

Azure AI Search documentation

Information retrieval at scale for agentic retrieval, with vector and text content in traditional or generative search scenarios.

About Azure AI Search

CONCEPT

[What is Azure AI Search?](#)

[AI enrichment](#)

[Semantic ranking](#)

HOW-TO GUIDE

[Create a search service](#)

[Configure a search service](#)

[Create a search index](#)

[Add vectors to an index](#)

[Query a vector index](#)

Agentic retrieval and vectors

CONCEPT

[Retrieval Augmented Generation \(RAG\)](#)

[Agentic retrieval](#)

[Vector search](#)

[Multimodal search](#)

[Built-in vectorization](#)

[Built-in compression](#)

QUICKSTART

[Classic RAG](#)

Code

GET STARTED

[Python samples](#)

[.NET samples](#)

[Vector samples ↗](#)

REFERENCE

[Azure REST API Reference](#)

[Azure SDK for .NET](#)

[Azure SDK for Python](#)

[Azure SDK for Java](#)

[Azure SDK for JavaScript](#)

What is Azure AI Search?

Azure AI Search is a fully managed, cloud-hosted service that connects your data to AI. The service unifies access to enterprise and web content so agents and LLMs can use context, chat history, and multi-source signals to produce reliable, grounded answers.

Common use cases include *classic search* and modern retrieval-augmented generation (RAG) via *agentic retrieval*. This makes Azure AI Search suitable for both enterprise and consumer scenarios, whether you're adding search functionality to a website, app, agent, or chatbot.

When you create a search service, you unlock the following capabilities:

- Two engines: [classic search](#) for single requests and [agentic retrieval](#) for parallel, iterative, LLM-assisted search.
- [Full-text](#), [vector](#), [hybrid](#), and [multimodal](#) queries over local (indexed) and remote content.
- AI enrichment to chunk, vectorize, and otherwise make raw content searchable.
- Relevance tuning to improve intent matching and result quality.
- Azure scale, security, monitoring, and compliance.
- Azure integrations with supported data platforms, Azure OpenAI, and Microsoft Foundry.

[Create a search service](#)

Why use Azure AI Search?

- Ground agents and chatbots in proprietary, enterprise, or web data for accurate, context-aware responses.
- Access data from Azure Blob Storage, Azure Cosmos DB, Microsoft SharePoint, Microsoft OneLake, and other supported data sources. Choose indexed or remote access based on your freshness, latency, and compliance needs.
- Enrich and structure content at indexing or query time with skills that perform chunking, embedding, and LLM-assisted transformations.
- Combine full-text search with vector search (hybrid search) to balance precision and recall.
- Query content containing both text and images in a single multimodal pipeline.
- Easily implement search-related features: relevance tuning, faceted navigation, filters (including geo-spatial search), synonym mapping, and autocomplete.
- Provide enterprise security, access control, and compliance through Microsoft Entra, Azure Private Link, document-level access control, and role-based access.

- Scale and operate in production with Azure reliability, monitoring and diagnostics (logs, metrics, and alerts), and REST API or SDK tooling for automation.

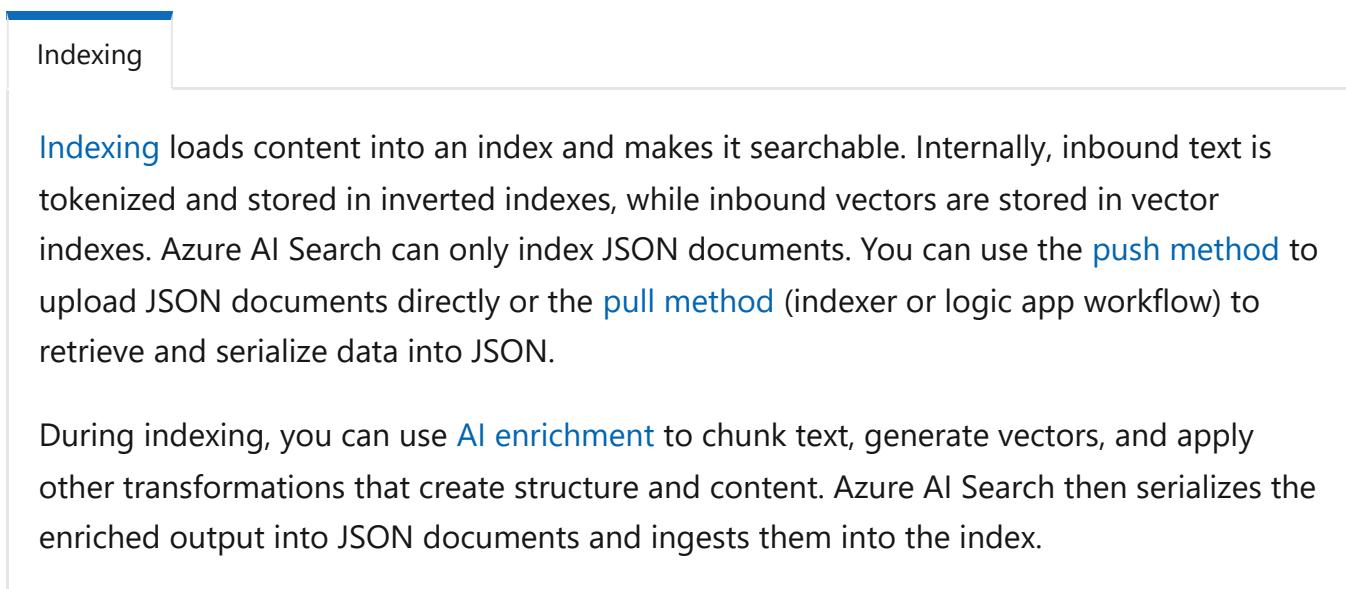
For more information about specific functionality, see [Features of Azure AI Search](#).

What is classic search?

Classic search is an index-first retrieval model for predictable, low-latency queries. Each query targets a single, predefined search index and returns ranked documents in one request-response cycle. No LLM-assisted planning, iteration, or synthesis occurs during retrieval.

In this architecture, your search service sits between the data stores that contain your unprocessed content and your client app. The app is responsible for sending query requests to your search service and handling the response.

This architecture has two primary workloads:



! Note

This diagram separates the indexing and query engines for clarity, but in Azure AI Search, they're the same component operating in read-write and read-only modes.

What is agentic retrieval?

[Agentic retrieval](#) is a multi-query pipeline designed for complex agent-to-agent workflows. Each query targets a [knowledge base](#) that represents a complete domain of knowledge. Your agent references the knowledge base for *what* to ground on, while the knowledge base handles *how* to perform grounding.

One knowledge base consists of one or more [knowledge sources](#), an optional LLM for query planning and answer synthesis, and parameters that govern retrieval behavior. Each query undergoes planning, decomposition into focused subqueries, parallel retrieval from knowledge sources, semantic reranking, and results merging. The three-pronged response is optimized for agent consumption.

Under the hood, agentic retrieval builds on the classic search architecture by adding a context layer (knowledge base) that orchestrates multi-source retrieval. Knowledge sources can be indexed or remote: indexed sources use the same indexing and query engines as classic search, while remote sources bypass indexing and are queried live.

How they compare

Classic search and agentic retrieval are complementary modes of information retrieval. Both support [full-text](#), [vector](#), [hybrid](#), and [multimodal](#) search. However, they differ in how content is ingested and queried. The following table summarizes their key differences.

Expand table

Aspect	Classic search	Agentic retrieval
Search corpus	Search index	Knowledge source
Search target	One index defined by a schema	A knowledge base pointing to one or more knowledge sources
Query plan	No plan, just a request	LLM-assisted or user-provided plan
Query request	Search documents in an index	Retrieve from knowledge sources
Response	Flattened search results based on schema	LLM-formulated answer or raw source data, activity log, references
Region restrictions	No	Yes
Status	Generally available	Public preview

How to get started

You can access Azure AI Search through the Azure portal, [REST APIs](#), and Azure SDKs for [.NET](#), [Java](#), [JavaScript](#), and [Python](#).

The portal is useful for service administration and content management, with tools for prototyping your knowledge bases, knowledge sources, indexes, indexers, skillsets, and data sources. REST APIs and SDKs are useful for production automation.

Choose your path

Before you get started, use this checklist to make key decisions:

- **Choose a search engine:** If you're not using an agent or chatbot, classic search can meet most app needs, with lower costs and complexity than LLM integration. If you want the benefits of a knowledge base and multiple knowledge sources without full LLM orchestration, consider agentic retrieval with the minimal [reasoning effort](#).
- **Choose a region:** If you're using agentic retrieval, choose a [supported region](#). For classic search, choose a region that offers the features and capacity you need.
- **Choose an ingestion method for index-bound content:** If your content is in a [supported data source](#), use the [pull method](#) to retrieve and serialize data into JSON. If you don't have a supported data source, or if your content and index must be synchronized in real time, the [push method](#) is your only option.
- **Do you need vectors?** LLMs and agents don't require vectors. Only use them if you need similarity search or if you have content that can be homogenized into vectors. Azure AI Search offers [integrated vectorization](#) for this task.
- **Do you need user-based permission inheritance?** Remote SharePoint is designed for this scenario, but you can also inherit user permissions attached to content in Azure Blob Storage or ADLS Gen2. For all other scenarios, you can use the [security filter](#) workaround.

Choose your learning resources

Quickstarts

We maintain quickstarts that span various end-to-end search scenarios:

- Quickstart: Agentic retrieval ([portal](#) or [programmatic](#))
- Quickstart: Full-text search ([portal](#) or [programmatic](#))

- Quickstart: Vector search ([portal](#) or [programmatic](#))

 Tip

For help with complex or custom solutions, [contact a partner](#) ↗ with deep expertise in Azure AI Search.

Last updated on 12/04/2025

What's new in Azure AI Search

Learn about the latest updates to Azure AI Search functionality, docs, and samples.

! Note

Preview features are announced here, but we also maintain a [preview features list](#) so you can find them in one place.

November 2025

[] Expand table

Item	Description
Search Service 2025-11-01-preview	New preview REST API version providing programmatic access to the data plane operations described in this table.
Semantic ranker and agentic retrieval on free tiers	Both semantic ranker and agentic retrieval are now available on free tiers in select regions , subject to limits on query volume.
Knowledge agents renamed to knowledge bases	Knowledge agents are now known as knowledge bases. Knowledge sources remain unchanged. This rename introduces breaking changes to REST API routes and properties. For help with migration, see Migrate your agentic retrieval code .
Knowledge bases (preview)	After being renamed from knowledge agents, knowledge bases now support new <code>retrievalInstructions</code> and <code>outputConfiguration</code> properties for improved query planning and execution control. It also provides a new reasoning effort for control over LLM processing.
Knowledge sources (preview)	New types of knowledge sources: indexed OneLake , indexed SharePoint , remote SharePoint , and web . For indexed knowledge sources, the new <code>ingestionParameters</code> object provides properties to control content ingestion and processing, including <code>contentExtractionMode</code> for use of the Azure Content Understanding skill and <code>ingestionPermissionOptions</code> for use of ACLs in the generated indexer.
Knowledge retrieval (preview)	Execute retrieval operations with support for reasoning effort , zero-model-call mode for efficiency, and partial responses.
Portal support for knowledge sources and knowledge bases (preview)	Use the Azure portal to create and manage knowledge sources and knowledge bases, with a new chat playground for sending retrieval requests. These portal-generated objects use the 2025-08-01-preview

Item	Description
	schema and have breaking changes with the 2025-11-01-preview. For help with migration, see Migrate your agentic retrieval code .
Foundry IQ (preview)	New integration that allows agents in Foundry Agent Service to invoke knowledge bases in Azure AI Search. Foundry IQ offloads complex retrieval operations to the knowledge base, enabling the agent to provide accurate, citation-backed responses based on enterprise data and web sources.
Azure Content Understanding skill (preview)	New skill that wraps Azure Content Understanding in Foundry Tools to extract structured Markdown from text, images, PDFs, Microsoft PowerPoint, Microsoft Word, and more. This skill provides advanced document parsing with better table extraction (including cross-page tables), image descriptions, and semantic chunking. For indexed knowledge sources, this skill is available through the <code>contentExtractionMode</code> property within <code>ingestionParameters</code> .
SharePoint indexer ACL support (preview)	Extended ACL support to flow basic SharePoint permissions to indexed documents, enabling document-level access control.
Elevated read permissions for ACLs (preview)	New capability to assign elevated read permissions to administrators for investigating problems with ACL configurations used in document access control.
Document-level sensitivity label indexing (preview)	New integration with Microsoft Purview to sync document sensitivity labels to the index, honoring their labels and protection at query time for data governance.
SharePoint indexing updates (preview)	New SharePoint indexer capabilities, including improved authentication options, incremental updates, and basic handling of document permissions.
Scoring function aggregation (preview)	New capability to combine and aggregate multiple scoring functions, enabling more sophisticated relevance customization and weighted signal combination.
Facet aggregations (preview)	New facet aggregation operations, including minimum, maximum, average, and cardinality, provide enhanced analytics in faceted search experiences.
<code>azure-api.net</code> endpoint support (preview)	The Azure OpenAI Embedding skill and Azure OpenAI vectorizer now accept <code>azure-api.net</code> endpoints for Azure API Management (not custom endpoints).
<code>services.ai.azure.com</code> endpoint support	The GenAI Prompt skill , Azure OpenAI Embedding skill , Azure OpenAI vectorizer , and AI enrichment now accept <code>services.ai.azure.com</code> endpoints for Microsoft Foundry resources. When you upgrade from Azure OpenAI to Foundry , a new project is automatically created and becomes available for RAG and multimodal RAG in the Import data (new) wizard .

September 2025

 Expand table

Item	Description
Search Service 2025-09-01	New stable REST API version supports general availability for Microsoft OneLake indexer, Document Layout skill, and other APIs.
Logic app workflow integration	Generally available.
OneLake indexer	Generally available.
Document Layout skill	Generally available.
Normalizers	Generally available.
Index description	Generally available.
Rescoring of binary quantized vectors	Generally available.
Rescoring options for scalar compressed vectors	Generally available.
Scoring profiles for semantically ranked results	Generally available.
Truncate dimensions	Generally available.
Unpack @search.score to view subscores in hybrid search results	Generally available.
Updates to import wizards (Phase 1)	<p>The Azure portal is undergoing a three-phase rollout to unify the two import wizards. For Phase 1, the Import and vectorize data wizard has been renamed to Import data (new) and redeveloped to support keyword search, modernizing the legacy Import data workflow with an improved interface and user experience.</p> <p>Import data (new) isn't a direct replacement for the old wizard. For example, it supports fewer skills for keyword search and doesn't offer built-in sample data.</p> <p>Both wizards are currently available, but Import data will be deprecated in a future phase.</p>

Item	Description
Support for Azure confidential computing	<p>Configure confidential computing during service creation to process data in use on confidential VMs. This compute type isn't intended for general use, but rather for stringent regulatory, compliance, or security requirements.</p>

August 2025

[Expand table](#)

Item	Description
Search Service 2025-08-01-preview	<p>New preview REST API version providing programmatic access to the data plane operations described in this table.</p>
Knowledge agents (preview)	<p>Architectural changes to the knowledge agent definition, which now requires one or more <code>knowledgeSources</code> instead of <code>targetIndexes</code> and deprecates <code>defaultMaxDocsForReranker</code>. New <code>retrievalInstructions</code> and <code>outputConfiguration</code> properties provide greater control over query planning and execution. For help with breaking changes, see Migrate your agentic retrieval code.</p>
Knowledge sources (preview)	<p>New REST APIs for creating and managing knowledge sources, which represent content that knowledge agents use to ground and answer queries. In this preview, you can create knowledge sources for search indexes and Azure blobs.</p>
Answer synthesis (preview)	<p>New <code>answerSynthesis</code> modality for knowledge agents. When specified, an LLM generates a natural-language answer as an embedded step in the retrieval pipeline. This differs from the default <code>extractiveData</code> modality, which returns raw search results for downstream processing.</p>
"Fast path" for knowledge agents (preview)	<p>(Removed in the 2025-11-01-preview. This documentation no longer exists). The <code>attemptFastPath</code> boolean in knowledge agents enabled a shorter processing time if queries are concise and the initial response is sufficiently relevant. Replacement feature is the minimal retrieval reasoning effort.</p>
Retrieval instructions (preview)	<p>New <code>retrievalInstructions</code> property for knowledge agents guides query planning in an agentic retrieval workflow. For example, you can specify criteria for including or excluding specific knowledge sources.</p>
Improved indexer runtime tracking	<p>Applies to Standard 3 High Density (S3 HD) services only. Get Service Statistics response now provides cumulative indexer processing information for the entire</p>

Item	Description
information (preview)	service. Get Status - Indexers provides the same information, but for a specific indexer.
Strict postfiltering for vector queries (preview)	New <code>strictPostFilter</code> mode for the <code>vectorFilterMode</code> parameter. When specified, filters are applied after the global top- <code>k</code> vector results are identified, ensuring that returned documents are a subset of the unfiltered results.
Increased maximum dimensions for vector fields	The maximum dimensions per vector field are now <code>4096</code> . This update applies to all stable and preview REST API versions that support vectors and doesn't introduce breaking changes.

July 2025

[] [Expand table](#)

Item	Description
Search Management 2025-05-01	Stable release of the REST APIs for the control plane operations described in this table. For migration guidance, see Upgrade to the latest REST API in Azure AI Search .
Service upgrade	Now generally available through Upgrade Service (REST API) and the Azure portal.
Pricing tier change	Now generally available through the <code>sku</code> property in Update Service (REST API) and the Azure portal.
User-assigned managed identity assignment	Now generally available through the <code>identity</code> property that associates a user-assigned managed identity with a search service configuration. Only the assignment step, via the Update Service (REST API) or the Azure portal, is generally available. APIs used for data source or model connections that include a user-assigned managed identity are still in preview.
Network security perimeter	Now generally available through the Azure Virtual Network Manager REST APIs , which are used to join a search service, and the Search Management REST APIs , which are used to view and synchronize the configuration settings. Portal support for both steps is also generally available.

May 2025

[] [Expand table](#)

Item	Description
Agentic retrieval (preview)	Create a conversational search experience powered by large language models (LLMs) and your proprietary data. Agentic retrieval breaks down complex user queries into subqueries, runs the subqueries in parallel, and extracts grounding data from documents indexed in Azure AI Search. The output is intended for agents and custom chat solutions. A new knowledge agent object is introduced in this preview. Its response payload is designed for downstream agent and chat model consumption, with full transparency of the query plan and reference data. To get started in the portal, see Quickstart: Agentic retrieval .
Multivector support (preview)	Index multiple child vectors within a single document field. You can now use vector types in nested fields of complex collections, effectively allowing multiple vectors to be associated with a single document.
Scoring profiles with semantic ranking (preview)	Semantic ranker adds a new field, <code>@search.rerankerBoostedScore</code> , to help you maintain consistent relevance and greater control over final ranking outcomes in your search pipeline.
Azure Logic Apps integration (preview)	Create an automated indexing pipeline that retrieves content using a logic app workflow. Use the Import and vectorize data wizard in the Azure portal to build an indexing pipeline based on Azure Logic Apps integration.
Document-level access control (preview)	Flow document-level permissions from blobs in Azure Data Lake Storage (ADLS) Gen2 to searchable documents in an index. Queries can now filter results based on user identity for selected data sources.
Multimodal search (preview)	Ingest, understand, and retrieve documents that contain text and images, enabling you to perform searches that combine various modalities, such as querying with text to find information embedded in relevant complex images. See Quickstart: Search for multimodal content for portal wizard support and Azure AI Search Multimodal RAG Demo for a code-first approach.
GenAI prompt skill (preview)	A new skill that connects to a large language model (LLM) for information, using a prompt you provide. With this skill, you can populate a searchable field using content from an LLM. A primary use case for this skill is <i>image verbalization</i> , using an LLM to describe images and send the description to a searchable field in your index.
Document Layout skill (preview)	New parameters are available for this skill if you use the 2025-05-01-preview API version or later. New parameters support image offset metadata that improves the image search experience.
Import and vectorize data wizard enhancements	This wizard provides two paths for creating and populating vector indexes: Retrieval Augmented Generation (RAG) and Multimodal RAG . Logic apps integration is through the RAG path.
Index "description" support (preview)	The latest preview API adds a description to an index. Consider a Model Context Protocol (MCP) server that must pick the correct index at run time. The decision can be based on the description rather than on the index name alone. The description must be human readable and under four thousand characters.

Item	Description
2025-05-01-preview	New data plane preview REST API version providing programmatic access to the preview features announced in this release.

April 2025

[+] [Expand table](#)

Item	Description
RAG Time Journey ↗	Code and video demonstrations of Retrieval Augmented Generation (RAG) workflows that use Azure AI Search. Segments include fundamentals, patterns and use-cases, vector indexing at scale ↗ , and agentic search ↗ where you use an agent to evaluate a result and generate a better answer.

March 2025

[+] [Expand table](#)

Item	Description
Service upgrade (preview)	Upgrade your search service to higher storage limits in your region. With a one-time upgrade, you no longer need to recreate your service. Available in Upgrade Service (2025-02-01-preview) and the Azure portal.
Pricing tier change (preview)	Change the pricing tier of your search service. This provides flexibility to scale storage, increase request throughput, and decrease latency based on your needs. Initially, this preview only supported upgrades between Basic and Standard (S1, S2, and S3) tiers, but starting in July 2025, it supports upgrades <i>and</i> downgrades between these tiers. Available in Update Service (2025-02-01-preview) and the Azure portal.
Facet hierarchies, aggregations, and facet filters (preview)	New facet query parameters support nested facets. For numeric facetable fields, you can sum the values of each field. You can also specify filters on a facet to add inclusion or exclusion criteria. Available in Search Documents (2025-03-01-preview) and the Azure portal.
Rescore vector queries over binary quantization using full precision vectors (preview)	For vector indexes that contain binary quantization, you can rescore query results using a full precision vector query. The query engine uses the dot product of the binary embeddings and the vector query for rescaling, which improves the quality of search results. Set <code>enableRescoring</code> and <code>discardOriginals</code> to use this feature, and call the latest preview API version on the request.

Item	Description
Semantic ranker prerelease models (preview)	Opt in to use prerelease semantic ranker models if one happens to be available in your region. Available in Create or Update Index (2025-03-01-preview) .
Search Service REST 2025-03-01-preview	Public preview release of REST APIs for data plane operations. Adds support for multi-vector embeddings, hierarchical facets, facet aggregation, and facet filters.
Search Management 2025-02-01-preview	Public review release of REST APIs for control plane operations. Adds support for in-place upgrade to higher capacity partitions, in-place upgrade to higher tiers, and Azure Confidential Compute.

February 2025

[+] [Expand table](#)

Item	Description
Customer-managed keys support for Managed HSM	Use either Azure Key Vault or Azure Key Vault Managed HSM (Hardware Security Module) to store customer-managed keys for extra encryption of sensitive content.

2024 announcements

[+] [Expand table](#)

Month	Type	Announcement
December	Template	RAG chat with Azure AI Search + Python . An AI application template for building a RAG solution using Azure AI Search and Python.
November	Security	Network security perimeter . Join a search service to a network security perimeter to control network access to your search service. The Azure portal and the Management REST APIs in the 2024-06-01-preview can be used to view and reconcile network security perimeter configurations.
November	Security	Shared private link support for Azure AI service connections . Connections to Azure AI for built-in skills processing can now be private using a shared private link on the connection.
November	Relevance	Rescoring options for compressed vectors . You can set options to rescore with original vectors instead of compressed vectors. Applies to HNSW and exhaustive KNN vector algorithms, using binary and scalar compression.

Month	Type	Announcement
		Available in the Create or Update Index (2024-11-01-preview) , the Azure portal, and in the Azure SDK beta packages that provide this feature.
November	Vector search	<p>Store fewer vector instances. In vector compression scenarios, you can omit storage of full precision vectors if you don't need them for rescore.</p> <p>Available in the Create or Update Index (2024-11-01-preview), the Azure portal, and in the Azure SDK beta packages that provide this feature.</p>
November	Relevance	<p>Query rewrite in the semantic reranker. You can set options on a semantic query to rewrite the query input into a revised or expanded query that generates more relevant results from the L2 ranker. Available in the Search Documents (2024-11-01-preview), the Azure portal, and in the Azure SDK beta packages that provide this feature.</p>
November	Relevance	<p>New semantic ranker models. Semantic ranker runs with improved models in all supported regions. There's no change to APIs or the Azure portal experience.</p>
November	Applied AI (skills)	<p>Document Layout skill. A new skill used to analyze a document for structure and provide structure-aware (paragraph) chunking. This skill calls Azure Document Intelligence in Foundry Tools and uses the Azure Document Intelligence layout model. Available in selected regions through the Create or Update Skillset (2024-11-01-preview), the Azure portal, and in the Azure SDK beta packages that provide this feature.</p>
November	Applied AI (skills)	<p>Keyless billing for Azure AI skills processing. You can now use a managed identity and roles for a keyless connection to Foundry Tools for built-in skills processing. This capability removes restrictions for having both search and Foundry Tools in the same region. Available in the Create or Update Skillset (2024-11-01-preview), the Azure portal, and in the Azure SDK beta packages that provide this feature.</p>
November	Indexer data source	<p>Markdown parsing mode. With this parsing mode, indexers can generate one-to-one or one-to-many search documents from Markdown files in Azure Storage and Microsoft OneLake. Available in the Create or Update Indexer (2024-11-01-preview), the Azure portal, and in the Azure SDK beta packages that provide this feature.</p>
November	API	<p>2024-11-01-preview. Preview release of REST APIs for query rewrite, Document Layout skill, keyless billing for skills processing, Markdown parsing mode, and rescore options for compressed vectors.</p>
November	Feature	<p>Portal support for structured data. The Import and vectorize data wizard now supports Azure SQL, Azure Cosmos DB, and Azure Table Storage.</p>
October	Feature	<p>Lower the dimension requirements for MRL-trained text embedding models on Azure OpenAI. Text-embedding-3-small and Text-embedding-3-large are trained using Matryoshka Representation Learning (MRL). This allows you to truncate the embedding vectors to fewer dimensions, and adjust the balance</p>

Month	Type	Announcement
		between vector index size usage and retrieval quality. A new <code>truncationDimension</code> in the 2024-09-01-preview enables access to MRL compression in text embedding models. This can only be configured for new vector fields.
October	Feature	Unpack @search.score to view subscores in hybrid search results . You can investigate Reciprocal Rank Fusion (RRF) ranked results by viewing the individual query subscores of the final merged and scored result. A new <code>debug</code> property unpacks the search score. <code>QueryResultDocumentSubscores</code> , <code>QueryResultDocumentRerankerInput</code> , and <code>QueryResultDocumentSemanticField</code> provide the extra detail. These definitions are available in the 2024-09-01-preview .
October	Feature	Target filters in a hybrid search to just the vector queries . A filter on a hybrid query involves all subqueries on the request, regardless of type. You can override the global filter to scope the filter to a specific subquery. The new <code>filterOverride</code> parameter is available on hybrid queries using the 2024-09-01-preview .
October	Applied AI (skills)	Text Split skill (token chunking) . This skill has new parameters that improve data chunking for embedding models. A new <code>unit</code> parameter lets you specify token chunking. You can now chunk by token length, setting the length to a value that makes sense for your embedding model. You can also specify the tokenizer and any tokens that shouldn't be split during data chunking. The new <code>unit</code> parameter and query subscore definitions are found in the 2024-09-01-preview .
October	API	2024-09-01-preview . Preview release of REST APIs for truncated dimensions in text-embedding-3 models, targeted vector filtering for hybrid queries, RRF subscore details for debugging, and token chunking for Text Split skill.
October	Feature	Portal support for customer-managed key encryption (CMK) . When you create new objects in the Azure portal, you can now specify CMK-encryption and select an Azure Key Vault to provide the key.
August	Feature	Debug Session improvements . There are two important improvements. First, you can now debug integrated vectorization and data chunking workloads. Second, Debug Sessions is redesigned for a more streamlined presentation of skills and mappings. You can select an object in the flow, and view or edit its details in a side panel. The previous tabbed layout is fully replaced with more context-sensitive information on the page.
August	API	2024-07-01 . Stable release of REST APIs for generally available vector data types, vector compression, and integrated vectorization during indexing and queries.
August	Feature	Integrated vectorization , Announcing general availability. Skills-driven data chunking and embedding during indexing.

Month	Type	Announcement
August	Feature	Vectorizers . Announcing general availability. Text-to-vector conversion during query execution. Both Azure OpenAI vectorizer and custom Web API vectorizer are generally available.
August	Feature	AzureOpenAIEmbedding skill . Announcing general availability. A skill type that calls an Azure OpenAI embedding model to generate embeddings during indexing.
August	Feature	Index projections . Announcing general availability. A component of a skillset definition that defines the shape of a secondary index, supporting a one-to-many index pattern, where content from an enrichment pipeline can target multiple indexes.
August	Feature	Binary and Scalar quantization . Announcing general availability. Compress vector index size in memory and on disk using built-in quantization.
August	Feature	Narrow data types . Announcing general availability. Assign a smaller data type on vector fields, assuming incoming data is of that data type.
August	Feature	Import and vectorize data wizard . Announcing general availability. A wizard that creates a full indexing pipeline that includes data chunking and vectorization. The wizard creates all necessary objects and configurations. This release adds wizard support for Azure Data Lake in Azure Storage.
August	Feature	stored property . Announcing general availability. Boolean that reduces storage of vector indexes by <i>not</i> storing retrievable vectors.
August	Feature	vectorQueries.Weight property . Announcing general availability. Specify the relative weight of each vector query in a search operation.
July	Accelerator	Chat with your data . A solution accelerator for the RAG pattern running in Azure, using Azure AI Search for retrieval and Azure OpenAI large language models to create conversational search experiences. The code with sample data is available for use case scenarios such as financial advisor and contract review and summarization.
July	Accelerator	Conversational Knowledge Mining . A solution accelerator built on top of Azure AI Search, Azure Speech in Foundry Tools, and Azure OpenAI that allows customers to extract actionable insights from post-contact center conversations.
July	Accelerator	Build your own copilot . Create your own custom copilot solution that empowers Client Advisor to harness the power of generative AI across both structured and unstructured data. Help our customers to optimize daily tasks and foster better interactions with more clients.
June	Feature	Image search in the Azure portal . Search explorer now supports image search. In a vector index that contains vectorized image content, you can drop images into Search Explorer to query for a match.

Month	Type	Announcement
May	Service limits	<p>Higher capacity and more vector quota at every tier (same billing rate). For most regions, partition sizes are now even larger for Standard 2 (S2), Standard 3 (S3), and Standard 3 High Density (S3 HD) for services created after April 3, 2024. To get the larger partitions, create a new service in a region that provides newer infrastructure.</p> <p>Storage Optimized tiers (L1 and L2) also have more capacity. L1 and L2 customers must create a new service to benefit from the higher capacity. There's no in-place upgrade at this time.</p> <p>Extra capacity is now available in more regions: Germany North, Germany West Central, South Africa North, Switzerland West, and Azure Government (Texas, Arizona, and Virginia).</p>
May	Feature	<p>OneLake integration (preview). New indexer for OneLake files and OneLake shortcuts. If you use Microsoft OneLake for data access to Amazon Web Services (AWS) and Google data sources, use this indexer to import external data into a search index. This indexer is available through the Azure portal, the 2024-05-01-preview REST API, and Azure SDK beta packages.</p>
May	Feature	<p>Vector relevance</p> <p>hybrid query relevance. Four enhancements improve vector and hybrid search relevance.</p> <p>First, you can now set thresholds on vector search results to exclude low-scoring results.</p> <p>Second, changes in the query architecture apply scoring profiles at the end of the query pipeline for every query type. Document boosting is a common scoring profile, and it now works as expected on vector and hybrid queries.</p> <p>Third, you can set MaxTextRecallSize and countAndFacetMode in hybrid queries to control the quantity of BM25-ranked search results that flow into the hybrid ranking model.</p> <p>Fourth, for vector and hybrid search, you can weight a vector query to have boost or diminish its importance in a multiquery request.</p>
May	Feature	<p>Binary vectors support. <code>Collection(Edm.Byte)</code> is a new supported data type. This data type opens up integration with the Cohere v3 binary embedding models and custom binary quantization. Narrow data types lower the cost of large vector datasets. See Index binary data for vector search for more information.</p>
May	Skill	<p>Azure Vision multimodal embeddings skill (preview). New skill that's bound to the multimodal embeddings API of Azure Vision. You can generate embeddings for text or images during indexing. This skill is available through the Azure portal and the 2024-05-01-preview REST API.</p>

Month	Type	Announcement
May	Vectorizer	Azure Vision vectorizer (preview). New vectorizer connects to an Azure Vision resource using the multimodal embeddings API to generate embeddings at query time. This vectorizer is available through the Azure portal and the 2024-05-01-preview REST API .
May	Vectorizer	Microsoft Foundry model catalog vectorizer (preview). New vectorizer connects to an embedding model deployed from the Foundry model catalog . This vectorizer is available through the Azure portal and the 2024-05-01-preview REST API .
		How to implement integrated vectorization using models from Foundry.
May	Skill	AzureOpenAIEmbedding skill (preview) supports more models on Azure OpenAI. Now supports text-embedding-3-large and text-embedding-3-small, along with text-embedding-ada-002 from the previous update. New <code>dimensions</code> and <code>modelName</code> properties make it possible to specify the various embedding models on Azure OpenAI. Previously, the dimensions limits were fixed at 1,536 dimensions, applicable to text-embedding-ada-002 only. The updated skill is available through the Azure portal and the 2024-05-01-preview REST API .
May	Portal	Import and vectorize data wizard now supports OneLake indexers as a data source. For embeddings, it also supports connections to Azure Vision multimodal, Foundry model catalog, and more embedding models on Azure OpenAI. When adding a field to an index, you can choose a binary data type . Search explorer now defaults to 2024-05-01-preview and supports the new preview features for vector and hybrid queries.
May	API	2024-05-01-preview . New preview version of the Search REST APIs provides new skills and vectorizers, new binary data type, OneLake files indexer, and new query parameters for more relevant results. See Upgrade REST APIs if you have existing code written against the 2023-07-01-preview and need to migrate to this version.
May	API	Azure SDK beta packages. Review the changelogs of the following Azure SDK beta packages for new feature support: Azure SDK for Python , Azure SDK for .NET , Azure SDK for Java
May	Samples	Python code samples . New end-to-end samples demonstrate integration with Cohere Embed v3 , integration with OneLake and cloud data platforms on Google and AWS , and integration with Azure Vision multimodal APIs .
April	API	Security update addressing information disclosure . GET responses no longer return connection strings or keys. Applies to GET Skillset, GET Index,

Month	Type	Announcement
		and GET Indexer. This change helps protect your Azure assets integrated with AI Search from unauthorized access.
April	API	2024-03-01-preview Search REST API
April	API	2024-03-01-preview Management REST API
April	API	2023-07-01-preview deprecation announcement . This version is no longer supported as of July 8, 2024. Newer API versions have a different vector configuration. You should migrate to a newer version as soon as possible.
April	Service limits	Basic and Standard tiers offer more storage per partition, at the same per-partition billing rate. Extra capacity is subject to regional availability and applies to new search services created after April 3, 2024. Basic now supports up to three partitions and three replicas.
April	Service limits	Vector quotas are higher on new services created after April 3, 2024 in selected regions.
April	Feature	Vector quantization, narrow vector data types, and a new stored property (preview) . Collectively, these three features minimize storage and costs.
February	Feature	New dimension limits on vector fields. Maximum dimension limits are now 3072 , up from 2048 .

Previous year's announcements

- [2023 announcements](#)
- [2022 announcements](#)
- [2021 announcements](#)
- [2020 announcements](#)
- [2019 announcements](#)

Service rebrand

This service has had multiple names over the years. Here they are in reverse chronological order:

- **Azure AI Search** (November 2023) Renamed to align with Foundry Tools and customer expectations.
- **Azure Cognitive Search** (October 2019) Renamed to reflect the expanded (yet optional) use of cognitive skills and AI processing in service operations.
- **Azure Search** (March 2015) The original name.

Service updates

You can find [service update announcements](#) for Azure AI Search on the Azure website.

Feature rename

Semantic search was renamed to [semantic ranker](#) in November 2023 to better describe the feature, which provides L2 ranking of an existing result set.

Last updated on 11/18/2025

Features of Azure AI Search

08/13/2025

Azure AI Search provides information retrieval and uses optional AI integration to extract more value from text and vector content.

The following table summarizes features by category. There's feature parity in all Azure public, private, and sovereign clouds, but some features aren't supported in [specific regions](#) or [specific tiers](#).

ⓘ Note

Looking for preview features? See the [preview features list](#).

[+] Expand table

Category	Features
Data sources	<p>Search indexes can accept text from any source, provided it's submitted as a JSON document.</p> <p>Indexers are a feature that automates data import from supported data sources to extract searchable content in primary data stores. Indexers handle JSON serialization for you and most support some form of change and deletion detection. You can connect to a variety of data sources, including OneLake (preview), Azure SQL Database, Azure Cosmos DB, or Azure Blob storage.</p> <p>Logic Apps connectors (preview) give you access to a broader range of data sources, including data on other cloud platforms. This indexing and enrichment pipeline is created in Azure AI Search but managed in Azure Logic Apps.</p>
Hierarchical and nested data structures	<p>Complex types and collections allow you to model virtually any type of JSON structure within a search index. One-to-many and many-to-many cardinality can be expressed natively through collections, complex types, and collections of complex types.</p>
Linguistic analysis	Analyzers are components used for text processing during indexing and search operations. By default, you can use the general-purpose Standard Lucene analyzer, or override the default with a language analyzer, a custom analyzer that you configure, or another predefined analyzer that

Category	Features
	<p>produces tokens in the format you require.</p> <p>Language analyzers from Lucene or Microsoft are used to intelligently handle language-specific linguistics including verb tenses, gender, irregular plural nouns (for example, 'mouse' vs. 'mice'), word decompounding, word-breaking (for languages with no spaces), and more.</p> <p>Custom lexical analyzers are used for complex query forms such as phonetic matching and regular expressions.</p>

Chat model and agent integration

[] [Expand table](#)

Category	Features
Chat completion models used during indexing	<p>GenAI prompt skill (preview) is a skill that calls a large language model during indexing and provides a prompt that determines the task. You decide what the task is. It might describing an image, summarizing or manipulating content, or any task the model can perform. Output is added as a new field in a searchable index.</p>
Chat completion models used at query time	<p>Agentic retrieval (preview) uses a large language model for query planning, decomposing and paraphrasing complex queries for better query coverage over your index. Responses from agentic retrieval are designed for agent-to-agent workflows. You can pass search results as single large string, which simplifies agent consumption of your proprietary content. The response also includes citations and query execution information.</p>
	<p>RAG patterns can be implemented using existing capabilities. The ability to tune for relevance and construct hybrid queries improve the quality of the content sent to chat bots for answer generation.</p>

Applied AI and AI enriched content

[] [Expand table](#)

Category	Features
AI processing during indexing	<p>AI enrichment refers to embedded image and natural language processing in an indexer pipeline that extracts text and information</p>

Category	Features
	<p>from content that can't otherwise be indexed for full text search. AI processing is achieved by adding and combining skills in a skillset, which is then attached to an indexer. AI can be either built-in skills from Microsoft, such as text translation or Optical Character Recognition (OCR), or custom skills that you provide.</p> <p>Integrated data chunking and vectorization splits up larger passages into smaller chunks that can be vectorized, with vectors routed to dedicated fields in an index for vector and hybrid search.</p>
AI processing during query execution	<p>Vectorizers are used to encode user query strings into vectors for vector search. You can use the same embedding models for queries that you used for indexing.</p>
Storing enriched content for analysis and consumption in non-search scenarios	<p>Knowledge store is persistent storage of AI enriched or AI generated content, intended for non-search scenarios like knowledge mining and data science workloads. A knowledge store is defined in a skillset, but created in Azure Storage as objects or tabular rowsets.</p>
Cached enrichments	<p>Enrichment caching (preview) refers to cached enrichments that can be reused during skillset execution. Caching is valuable in skillsets that include OCR and image analysis, which are expensive to process.</p>

Vector and hybrid retrieval

[] [Expand table](#)

Category	Features
Vector indexing	<p>Within a search index, add vector fields to support vector search scenarios. Vector fields can coexist with nonvector fields in the same search document.</p>
Vector queries	<p>Formulate single and multiple vector queries.</p>
Vector search algorithms	<p>Use Hierarchical Navigable Small World (HNSW) or exhaustive K-Nearest Neighbors (KNN) to find similar vectors in a search index.</p>
Vector filters	<p>Apply filters before or after query execution for greater precision during information retrieval.</p>
Hybrid information retrieval	<p>Search for concepts and keywords in a single hybrid query request.</p> <p>Hybrid search consolidates vector and text search, with optional semantic ranking and relevance tuning for best results.</p>

Category	Features
Integrated data chunking and vectorization	<p>Native data chunking through Text Split skill. Native vectorization through vectorizers and embedding skills such as AzureOpenAIEmbeddingModel, Azure AI Vision multimodal, and the AML skill that you can use to connect to endpoints in the Azure AI Foundry model catalog.</p> <p>Integrated vectorization provides an end-to-end indexing pipeline from source files to queries.</p>
Integrated vector compression and quantization	<p>Use built-in scalar and binary quantization to reduce vector index size in memory and on disk. You can also forego storage of vectors you don't need, or assign narrow data types to vector fields for reduced storage requirements.</p>

Full text and other query forms

[] [Expand table](#)

Category	Features
Free-form text search	<p>Full-text search is a primary use case for most search-based apps. Queries can be formulated using a supported syntax.</p> <p>Simple query syntax provides logical operators, phrase search operators, suffix operators, precedence operators.</p> <p>Full Lucene query syntax includes all operations in simple syntax, with extensions for fuzzy search, proximity search, term boosting, and regular expressions.</p>
Relevance	<p>Simple scoring is a key benefit of Azure AI Search. Scoring profiles are used to model relevance as a function of values in the documents themselves. For example, you might want newer products or discounted products to appear higher in the search results. You can also build scoring profiles using tags for personalized scoring based on customer search preferences you've tracked and stored separately.</p> <p>Semantic ranker is premium feature that reranks results based on semantic relevance to the query. Depending on your content and scenario, it can significantly improve search relevance with almost minimal configuration or effort.</p>
Geospatial search	<p>Geospatial functions filter over and match on geographic coordinates. You can match on distance or by inclusion in a polygon shape.</p>

Category	Features
Filters and facets	<p>Faceted navigation is enabled through a single query parameter. Azure AI Search returns a faceted navigation structure you can use as the code behind a categories list, for self-directed filtering (for example, to filter catalog items by price-range or brand).</p> <p>Filters can be used to incorporate faceted navigation into your application's UI, enhance query formulation, and filter based on user- or developer-specified criteria. Create filters using the OData syntax.</p>
User experience	<p>Autocomplete can be enabled for type-ahead queries in a search bar.</p> <p>Search suggestions also works off of partial text inputs in a search bar, but the results are actual documents in your index rather than query terms.</p> <p>Synonyms associates equivalent terms that implicitly expand the scope of a query, without the user having to provide the alternate terms.</p> <p>Hit highlighting applies text formatting to a matching keyword in search results. You can choose which fields return highlighted snippets.</p> <p>Sorting is offered for multiple fields via the index schema and then toggled at query-time with a single search parameter.</p> <p>Paging and throttling your search results is straightforward with the finely tuned control that Azure AI Search offers over your search results.</p>

Security features

 Expand table

Category	Features
Network security	<p>IP rules for inbound firewall support allows you to set up IP ranges over which the search service accepts requests.</p> <p>Create a private endpoint using Azure Private Link to force all requests through a virtual network.</p> <p>Network security perimeter support allows you to join Azure AI Search to a network security perimeter that includes other Azure resources so that you can manage network access holistically.</p>
Data encryption	<p>Microsoft-managed encryption-at-rest is built into the internal storage layer and is irrevocable.</p>

Category	Features
	<p>Customer-managed encryption keys (CMK) that you create and manage in Azure Key Vault can be used for supplemental encryption of indexes and synonym maps. For services created after August 1 2020, CMK encryption extends to data on temporary disks, for full double encryption of indexed content.</p>
Inbound access	<p>Role-based access control assigns roles to users and groups in Microsoft Entra ID for controlled access to search content and operations. You can also use key-based authentication if you don't want to use role assignments.</p> <p>Document-level access control (preview) filters out search results that a user isn't authorized to see. For several data sources, if the data source provides an access control model, you can configure an index to inherit the user permission metadata.</p>
Outbound security (indexers)	<p>Data connections through private endpoints allows an indexer to connect to Azure resources that are protected through Azure Private Link.</p> <p>Data connections through managed identities authenticates connections to Azure resources using a Microsoft Entra security principal, which eliminates storage and passing of hardcoded API keys.</p> <p>Data access using a trusted identity means that connection strings to external data sources can omit user names and passwords. When an indexer connects to the data source, the resource allows the connection if the search service was previously registered as a trusted service (applies to Azure Storage only).</p>

Portal features

[] Expand table

Category	Features
Tools for prototyping and inspection	<p>Add index is an index designer in the Azure portal that you can use to create a basic schema consisting of attributed fields and a few other settings. After saving the index, you can populate it using an SDK or the REST API to provide the data.</p> <p>Import data wizard creates indexes, indexers, skillsets, and data source definitions. If your data exists in Azure, this wizard can save you significant time and effort, especially for proof-of-concept investigation and exploration.</p> <p>Import and vectorize data wizard creates a full indexing pipeline that</p>

Category	Features
	<p>includes data chunking and vectorization. The wizard creates all of the objects and configuration settings.</p> <p>Search explorer is used to test queries and refine scoring profiles.</p> <p>Create demo app is used to generate an HTML page that can be used to test the search experience.</p> <p>Debug Sessions is a visual editor that lets you debug a skillset interactively. It shows you dependencies, output, and transformations.</p>
Monitoring and diagnostics	<p>Enable monitoring features to go beyond the metrics-at-a-glance that are always visible in the Azure portal. Metrics on queries per second, latency, and throttling are captured and reported in portal pages with no extra configuration required.</p>

Programmability

 [Expand table](#)

Category	Features
REST	<p>Service REST API is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics.</p> <p>Management REST API is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.</p>
Azure SDK for .NET	<p>Azure.Search.Documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics.</p> <p>Microsoft.Azure.Management.Search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.</p>
Azure SDK for Java	<p>com.azure.search.documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics.</p> <p>com.microsoft.azure.management.search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.</p>

Category	Features
Azure SDK for Python	<p>azure-search-documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics.</p> <p>azure-mgmt-search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.</p>
Azure SDK for JavaScript/TypeScript	<p>azure/search-documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics.</p> <p>azure/arm-search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.</p>

Try Azure AI Search for free

If you're new to Azure, you can set up an Azure free account to explore Azure AI Search and other services at no charge. Information retrieval over your own content is useful for many scenarios including AI generative search.

This article explains how to get the most value from your Azure free account so that you can complete your evaluation of Azure AI Search quickly and efficiently.

Step one: Sign up for an Azure free account

To try Azure AI Search for free, [sign up for an Azure free account](#). The free account is active for 30 days, and comes with free credits so that you can create billable services at no charge.

Currently, the credit is equivalent to USD 200. The exact amount is subject to change, and you can verify the credit on the Azure sign-up page.

[Try Azure for free](#)

After you sign up, you can immediately use either of these links to access Azure resources and experiences:

- [Sign in to Azure portal](#) to view, manage, and create more resources. You can also use the Azure portal to track your credits and projected costs.
- [Sign in to the Microsoft Foundry portal](#) for a no-code approach to deploying models on Azure OpenAI and using Azure AI Search for information retrieval. **We recommend you start here first.**

Step two: "Day One" tasks

In the Foundry (new) portal, you can create an end-to-end solution that integrates Azure AI Search and Foundry Agent Service for knowledge retrieval.

The portal supports creating *knowledge sources* that map to your indexed content in Azure AI Search and *knowledge bases* that orchestrate retrieval operations, including query decomposition, hybrid search, and result reranking. You can then configure a Foundry agent to use this knowledge base as a Model Context Protocol (MCP) tool, allowing the agent to retrieve relevant information and provide grounded, citation-backed responses.

For more information about the programmatic experience, see [Connect a Foundry IQ knowledge base to Foundry Agent Service](#).

Step three: Have a plan for next steps

The trial period can go by quick. Having a plan of action can help you get the most out of your trial subscription. For Azure AI Search, most newer customers and developers are exploring RAG patterns.

For a next step evaluation of [RAG scenarios](#), you should have three or five Azure resources for:

- Storing data
- Deploying embedding and chat models ([Azure OpenAI](#))
- Applying Foundry Tools for creating AI-generated content during indexing (optional)
- Adding information retrieval ([Azure AI Search](#))
- Adding a frontend app (optional)

Many of our quickstarts and tutorials use Azure Storage, so we recommend creating an Azure Storage account for getting started.

Generative search requires embedding and chat models. The Azure cloud provides Azure OpenAI, but you can also use Azure Vision in Foundry Tools for multimodal embeddings (but not chat). Another model provider is Foundry and deploying chat and embedding models into the model catalog. However, for initial exploration, we recommend Azure OpenAI for its familiarity and mainstream offerings.

Application frontends are useful if you're prototyping a solution for a wider audience. You can use Azure Web apps or build an ASP.NET MVC application for this task. Otherwise, if you're working locally, you can view output in Jupyter notebooks in Visual Studio Code or another IDE. Or view results in console apps or other apps that run on localhost.

Check regions

Azure AI Search offers integrated operations with applied AI in the Azure cloud. For data residency and efficient operations, integration typically depends on services running within the same region.

Note

The same-region requirement doesn't apply to Azure OpenAI and Foundry for interoperability with Azure AI Search. However, using the same region can improve performance and reduce latency.

For [AI enrichment](#), [integrated vectorization](#), and [multimodal search](#) powered by Foundry Tools, you must create Azure AI Search and Foundry in the same region. This is required for [billing](#)

purposes.

Before you create these resources:

- Check [Azure AI Search regions](#). The **AI enrichment** column indicates whether Azure AI Search and Foundry are in the same region.
- Check [Azure Vision regions](#). The **Multimodal embeddings** column indicates regional support for the multimodal APIs that enable text and image vectorization. Azure Vision provides these APIs, which you access through a Foundry resource. Ensure that your search service and Foundry resource are in the same region as the multimodal APIs.

Create services

1. [Create a search service](#) if you don't have one already. Choose the Basic tier and, if applicable, the same region as Foundry. Most Azure AI Search regions provide higher capacity storage limits. There are just a few that have older and lower limits. For the Basic tier, as you install, confirm that you have a 15-GB partition.
[Create a search service](#)
2. [Create an Azure Storage account](#). Choose a general purpose account and use default settings.
3. [Create an Azure OpenAI resource](#).
4. [Create a Foundry resource](#) to use applied AI in your indexing workloads and Azure Vision multimodal APIs as an embedding model provider. You can create and transform content during indexing if applied AI can be attached. For multimodal APIs, make sure you choose a region that provides those APIs. Look for this tile in Azure Marketplace:

Try the quickstarts

Try the Azure portal quickstarts for Azure AI Search or quickstarts that use Visual Studio Code with REST or Python extensions. It's the fastest approach creating searchable content, and you don't need coding skills to complete the tasks.

- [Quickstart: Vector search in the Azure portal](#)
- [Quickstart: Image search in the Azure portal](#)
- [Quickstart: Keyword in the Azure portal](#)
- [Quickstart: Generative search \(RAG\) using a Python client](#)
- [Quickstart: Vector search using a REST client](#)

Foundry supports connecting to content in Azure AI Search.

- [Quickstart: Chat using your own data with Azure OpenAI](#)
- [Tutorial: Build a custom chat app with the prompt flow SDK](#)

Developers should review [azure-search-vector-samples](#) repository or the solution accelerators. You can deploy and run any of these samples using the Azure trial subscription.

Many samples and [accelerators](#) come with bicep scripts that deploy all Azure resources and dependencies, so you can skip installation steps and explore an operational solution as soon as the development completes.

Step four: Track your credits

During the trial period, you want to stay under the USD 200 credit allocation. Most services are Standard, so you won't be charged while they're not in use, but an Azure AI Search service on the Basic tier is provisioned on dedicated clusters and it can only be used by you. It's billable during its lifetime. If you provision a basic search service, expect Azure AI Search to consume about one third of your available credits during the trial period.

During the trial period, the Azure portal provides a notification on the top right that tells you how many credits are used up and what remains.

You can also monitor billing by searching for *subscriptions* in the Azure portal to view subscription information at any time. The Overview page gives you spending rates, forecasts, and cost management. For more information, see [Check usage of free services included with your Azure free account](#).

Consider the free tier

You can create a search service that doesn't consume credits. Here are some points about the free tier to keep in mind:

- You can have one free search service per Azure subscription.
- You can complete all of the quickstarts and most tutorials, except for those featuring semantic ranking and managed identities for Microsoft Entra ID authentication and authorization.
- Storage is capped at 50 MB.
- You can have up to three indexes, indexers, data sources, and skillset at one time.

Review the [service limits](#) for other constraints that apply to the free tier.

 Note

Free services that remain inactive for an extended period of time might be deleted to free up capacity if the region is under capacity constraints.

Next steps

Sign up for an Azure free account:

[Try Azure for free](#)

When you're ready, add Azure AI Search as your first resource:

[Create a search service](#)

Last updated on 11/18/2025

Azure AI Search Frequently Asked Questions

Find answers to commonly asked questions about Azure AI Search.

General

What is Azure AI Search?

Azure AI Search provides a dedicated search engine and persistent storage of your searchable content for agentic, full-text, and vector search scenarios. It also includes optional integrated AI to extract text and structure from raw content and to chunk and vectorize content for vector search.

How do I work with Azure AI Search?

The primary workflow is create, load, and query an index. Although you can use the Azure portal for most tasks, Azure AI Search is intended to be used programmatically, handling requests from client code. Programmatic support is provided through REST APIs and client libraries in .NET, Python, Java, and JavaScript SDKs for Azure.

Are "Azure Search," "Azure Cognitive Search," and "Azure AI Search" the same product?

Yes. They're all the same product, with rebranding occurring in October 2019 and again in October 2023. You might occasionally see evidence of the former names at the programmatic level.

What languages are supported?

For vectors, the embedding models you use determines the linguistic experience.

For nonvector strings and numbers, the default analyzer used for tokenization is standard Lucene, which is language agnostic. Otherwise, language support is expressed through [language analyzers](#) that apply linguistic rules to inbound (indexing) and outbound (queries) content. Some features, such as [speller](#) and [query rewrite](#), are limited to a subset of languages.

How do I integrate search into my solution?

Client code should call the Azure SDK client libraries or REST APIs to connect to a search index, formulate queries, and handle responses. You can also write code that builds and refreshes an index, or that runs indexers programmatically or via script.

Can I pause the service and stop billing?

You can't pause a search service. In Azure AI Search, computing resources are allocated when the service is created. It's not possible to release and reclaim those resources on demand.

Can I upgrade or downgrade the service?

Services created before April 2024 in select regions can be [upgraded to higher-capacity clusters](#). Downgrading your service isn't supported.

To get more or less capacity, you can also [switch to a different pricing tier](#). Scaling is blocked if your current service configuration exceeds the [limits of the target tier](#) or if your region has [capacity constraints on the target tier](#). Currently, you can only switch between Basic and Standard (S1, S2, and S3) tiers.

Can I rename or move the service?

Service name and region are fixed for the lifetime of the service.

If I migrate my search service to another subscription or resource group, should I expect any downtime?

As long as you follow the [checklist before moving resources](#) and make sure each step is completed, there shouldn't be any downtime.

Why do I see different storage limits for same-tier search services?

Storage limits can vary by service creation date. In most supported regions, [newer services have higher storage limits than older services](#), even if they're on the same tier. However, you might be able to [upgrade your old service](#) to access the new limits.

Indexing

What does "indexing" mean in Azure AI Search?

It refers to the ingestion, parsing, and storing of textual content and tokens that populate a search index. Indexing creates inverted indexes and other physical data structures that support information retrieval.

It creates vector indexes if the schema includes vector fields.

Can I move, backup, and restore indexes?

There's no native support for porting indexes. Search indexes are considered downstream data structures, accepting content from other data sources that collect operational data. Therefore, there's no built-in support for backing up and restoring indexes. The expectation is that you would rebuild an index from source data if it was deleted or needed to be moved.

However, if you want to move an index between search services, you can try the backup and restore code sample for [.NET ↗](#) or [Python ↗](#).

Can I restore my index or service after it's deleted?

No. If you delete an Azure AI Search index or service, it can't be recovered. When you delete a search service, all indexes in the service are permanently deleted.

Can I index from SQL Database replicas?

If you're using the search indexer for Azure SQL Database, there are no restrictions on the use of primary or secondary replicas as a data source when building an index from scratch. However, refreshing an index with incremental updates (based on changed records) requires the primary replica. This requirement comes from SQL Database, which guarantees change tracking on primary replicas only. If you try using secondary replicas for an index refresh workload, there's no guarantee you get all of the data.

Vectors

What is vector search?

Vector search is a technique that finds the most similar documents by comparing their vector representations. Because the goal of a vector representation is to capture the essential characteristics of an item in a numerical format, vector queries can identify similar content, even if there are no explicit matches based on keywords or tags.

When a user performs a search, the query is summarized into a vector representation, and the vector search engine identifies the most similar documents. To improve efficiency on large databases, vector search often provides the Approximate Nearest Neighbors (ANN) for a query vector. For more information, see [Vector search in Azure AI Search](#).

Does Azure AI Search support vector search?

Azure AI Search supports vector indexing and retrieval. It can chunk and vectorize query strings and content if you use [integrated vectorization](#), which takes a dependency on indexers and skillsets.

How does vector search work in Azure AI Search?

With standalone vector search, you first use an embedding model to transform content into a vector representation within an embedding space. You can then provide these vectors in a document payload to the search index for indexing. To serve search requests, you use the same embedding model to transform the search query into a vector representation, and vector search finds the most similar vectors and return the corresponding documents.

In Azure AI Search, you can index vector data as fields in documents alongside textual and other types of content. There are [multiple data types](#) for vector fields.

Vector queries can be issued standalone or in combination with other query types, including term queries and filters in the same search request.

Can Azure AI Search vectorize my content or queries?

[Built-in integrated vectorization](#) is now generally available.

Does my search service support vector search?

Most existing services support vector search. If you're using a package or API that supports vector search and index creation fails, the underlying search service doesn't support vector search, and a new service must be created. This can occur for a small subset of services created prior to January 1, 2019.

Can I add vector search to an existing index?

If your search service supports vector search, both existing and new indexes can accommodate vector fields.

Why do I see different vector index size limits between my new search services and existing search services?

Azure AI Search rolled out improved vector index size limits worldwide for new search services, but [some regions experience capacity constraints](#), and some regions don't have the required infrastructure. New search services created after May 2024 in supported regions should see increased vector index size limits. Alternatively, if you have an existing service in a supported region, you might be able to [upgrade your service](#) to access the new limits.

Why does my vector index show zero storage?

Only vector indexes that use the Hierarchical Navigable Small World (HNSW) algorithm report on vector index size in the Azure portal. If your index uses exhaustive KNN, vector index size is reported as zero, even though the index contains vectors.

How do I enable vector search on a search index?

To enable vector search in an index:

- Add one or more vector fields to a field collection.
- Add a "vectorSearch" section to the index schema specifying the configuration used by vector search fields, including the parameters of the ANN algorithm used, like HNSW.
- Use the latest stable version, [2025-09-01](#), or an Azure SDK to create or update the index, load documents, and issue queries. For more information, see [Create a vector index](#).

Queries

Where does query execution occur?

Queries execute over a single search index that's hosted on your search service. You can't join multiple indexes to search content in two or more indexes, but you can [query same-name indexes in multiple search services](#).

Why are there zero matches on terms I know to be valid?

The most common case isn't knowing that each query type supports different search behaviors and levels of linguistic analyses. Full-text search, which is the predominant workload, includes a language analysis phase that breaks down terms to root forms. This aspect of query parsing casts a broader net over possible matches, because the tokenized term matches a greater number of variants.

However, wildcard, fuzzy, and regex queries aren't analyzed like regular term or phrase queries and can lead to poor recall if the query doesn't match the analyzed form of the word in the search index. For more information about query parsing and analysis, see [Full-text search in Azure AI Search](#).

Why are my wildcard searches slow?

Most wildcard search queries, like prefix, fuzzy, and regex, are rewritten internally with matching terms in the search index. This extra processing adds to latency. Additionally, broad search queries like `a*` are likely to be rewritten with many terms, which can be slow. For performant wildcard searches, consider defining a [custom analyzer](#).

Can I search across multiple indexes?

No. A query is always scoped to a single index.

Why is the search score a constant 1.0 for every match?

Search scores are generated for full-text search queries based on the [statistical properties of matching terms](#) and are ordered from high to low in the result set. Query types that aren't full-text search (wildcard, prefix, regex) aren't ranked by a relevance score. A constant score allows matches found through query expansion to be included in the results without affecting the ranking.

For example, suppose an input of "tour*" in a wildcard search produces matches on "tours", "tourettes", and "tourmaline". Given the nature of these results, there's no way to reasonably infer which terms are more valuable than others. For this reason, term frequencies are ignored when scoring results in queries of types wildcard, prefix, and regex. Search results based on a partial input are given a constant score to avoid bias towards potentially unexpected matches.

Security

Where does Azure AI Search store customer data?

It stores your data in the [geography \(Geo\)](#) where your service is deployed. Microsoft might replicate your data within the same geo for high availability and durability. For more information, see [Data residency in Azure](#).

Does Azure AI Search send customer data to other services for processing?

Yes. Skills and vectorizers make [outbound calls from Azure AI Search](#) to other Azure resources or external models that you specify for embedding or chat. Calls to those APIs typically contain raw content to be processed or queries that are vectorized by an embedding model. For Azure-to-Azure connections, the service sends requests over the internal network. If you add a custom skill or vectorizer, the indexer sends content to the URI provided in the custom skill over the public network unless you configure a [shared private link](#).

Does Azure AI Search process customer data in other regions?

Processing (vectorization or applied AI transformations) is performed in the Geo that hosts the Foundry Tools subservice used by skills, the Azure apps or functions hosting custom skills, or the Azure OpenAI or Microsoft Foundry region that hosts your deployed models. These resources are specified by you, so you can choose whether to deploy them in the same Geo as your search service or not.

If you send data to external (non-Azure) models or services, the processing location is determined by the external service.

Can I control access to search results based on user identity?

You can if you implement a solution that associates documents with a user identity. Typically, users who are authorized to run your application are also authorized to see all search results. Azure AI Search doesn't have built-in support for row-level or document-level permissions, but you can implement [security filters](#) as a workaround. For steps and script, see [Get started: Chat using your own data \(Python sample\)](#).

Can I control access to operations based on user identity?

Yes. You can use [role-based authorization](#) for data plane operations over content.

Can I use the Azure portal to view and manage search content if my search service is behind an IP firewall or private endpoint?

You can use the Azure portal on a network-protected search service if you create a network exception that allows client and portal access. For more information, see [Connect through an IP firewall](#) or [Connect through a private endpoint](#).

Next steps

If your question isn't answered here, you can refer to the following sources for more questions and answers.

[Stack Overflow: Azure AI Search ↗](#)

[How full-text search works in Azure AI Search](#)

[What is Azure AI Search?](#)

Data import in Azure AI Search

09/23/2025

In Azure AI Search, queries execute over your content that's loaded into a [search index](#). This article describes the two basic workflows for populating an index: *push* your data into the index programmatically, or *pull* in the data using a [search indexer](#).

Both approaches load documents from an external data source. Although you can create an empty index, it's not queryable until you add the content.

(!) Note

If [AI enrichment](#) or [integrated vectorization](#) are solution requirements, you must use the pull model (indexers) to load an index. Skillsets are attached to indexers and don't run independently.

Pushing data to an index

Push model is an approach that uses APIs to upload documents into an existing search index. You can upload documents individually or in batches up to 1000 per batch, or 16 MB per batch, whichever limit comes first.

Key benefits include:

- No restrictions on data source type. The payload must be composed of JSON documents that map to your index schema, but the data can be sourced from anywhere.
- No restrictions on frequency of execution. You can push changes to an index as often as you like. For applications having low latency requirements (for example, when the index needs to be in sync with product inventory fluctuations), the push model is your only option.
- Connectivity and the secure retrieval of documents are fully under your control. In contrast, indexer connections are authenticated using the security features provided in Azure AI Search.

How to push data to an Azure AI Search index

Use the following APIs to load single or multiple documents into an index:

- [Index Documents \(REST API\)](#)

- [IndexDocumentsAsync](#) (Azure SDK for .NET) or [SearchIndexingBufferedSender](#)
- [IndexDocumentsBatch](#) (Azure SDK for Python) or [SearchIndexingBufferedSender](#)
- [IndexDocumentsBatch](#) (Azure SDK for Java) or [SearchIndexingBufferedSender](#)
- [IndexDocumentsBatch](#) (Azure SDK for JavaScript) or [SearchIndexingBufferedSender](#)

There's no support for pushing data via the Azure portal.

For an introduction to the push APIs, see:

- [Quickstart: Full-text search](#)
- [C# Tutorial: Optimize indexing with the push API](#)
- [REST Quickstart: Create an Azure AI Search index using PowerShell](#)

Indexing actions: upload, merge, mergeOrUpload, delete

You can control the type of indexing action on a per-document basis, specifying whether the document should be uploaded in full, merged with existing document content, or deleted.

Whether you use the REST API or an Azure SDK, the following document operations are supported for data import:

- **Upload**, similar to an "upsert" where the document is inserted if it's new, and updated or replaced if it exists. If the document is missing values that the index requires, the document field's value is set to null.
- **merge** updates a document that already exists, and fails a document that can't be found. Merge replaces existing values. For this reason, be sure to check for collection fields that contain multiple values, such as fields of type `Collection(Edm.String)`. For example, if a `tags` field starts with a value of `["budget"]` and you execute a merge with `["economy", "pool"]`, the final value of the `tags` field is `["economy", "pool"]`. It won't be `["budget", "economy", "pool"]`.
- **mergeOrUpload** behaves like **merge** if the document exists, and **upload** if the document is new.
- **delete** removes the entire document from the index. If you want to remove an individual field, use **merge** instead, setting the field in question to null.

Pulling data into an index

The pull model uses *indexers* connecting to a supported data source, automatically uploading the data into your index. Indexers from Microsoft are available for these platforms:

- Azure Blob storage
- Azure Table storage
- Azure Data Lake Storage Gen2
- Azure Files (preview)
- Azure Cosmos DB
- Azure SQL Database, SQL Managed Instance, and SQL Server on Azure VMs
- Microsoft OneLake files and shortcuts
- SharePoint Online (preview)

You can use third-party connectors, developed and maintained by Microsoft partners. For more information and links, see [Data source gallery](#).

Indexers connect an index to a data source (usually a table, view, or equivalent structure), and map source fields to equivalent fields in the index. During execution, the rowset is automatically transformed to JSON and loaded into the specified index. All indexers support schedules so that you can specify how frequently the data is to be refreshed. Most indexers provide change tracking if the data source supports it. By tracking changes and deletes to existing documents in addition to recognizing new documents, indexers remove the need to actively manage the data in your index.

How to pull data into an Azure AI Search index

Use the following tools and APIs for indexer-based indexing:

- Azure portal: [Import wizards](#)
- REST APIs: [Create Indexer \(REST\)](#), [Create Data Source \(REST\)](#), [Create Index \(REST\)](#)
- Azure SDK for .NET: [SearchIndexer](#), [SearchIndexerDataSourceConnection](#), [SearchIndex](#),
- Azure SDK for Python: [SearchIndexer](#), [SearchIndexerDataSourceConnection](#), [SearchIndex](#),
- Azure SDK for Java: [SearchIndexer](#), [SearchIndexerDataSourceConnection](#), [SearchIndex](#),
- Azure SDK for JavaScript: [SearchIndexer](#), [SearchIndexerDataSourceConnection](#),
[SearchIndex](#),

Indexer functionality is exposed in the Azure portal, the [REST API](#), and the [.NET SDK](#).

An advantage to using the Azure portal is that Azure AI Search can usually generate a default index schema by reading the metadata of the source dataset.

Verify data import with Search explorer

A quick way to perform a preliminary check on the document upload is to use [Search explorer](#) in the Azure portal.

The explorer lets you query an index without having to write any code. The search experience is based on default settings, such as the [simple syntax](#) and default [searchMode query parameter](#). Results are returned in JSON so that you can inspect the entire document.

Here's an example query that you can run in Search Explorer in JSON view. The "HotelId" is the document key of the hotels-sample-index. The filter provides the document ID of a specific document:

```
JSON

{
  "search": "*",
  "filter": "HotelId eq '50'"
}
```

If you're using REST, this [Look up query](#) achieves the same purpose.

See also

- [Indexer overview](#)
- [Portal quickstart: create, load, query an index](#)

Search indexes in Azure AI Search

In Azure AI Search, a *search index* is your searchable content, available to the search engine for indexing, agentic search, full-text search, vector search, hybrid search, and filtered queries. An index is defined by a schema and saved to the search service, with data ingestion following as a second step. Indexed content exists within your search service, apart from your primary external data stores, which is necessary for the millisecond response times expected in modern search applications. Except for indexer-driven indexing scenarios, the search service never connects to or queries your external source data.

This article covers the key concepts for creating and managing a search index, including:

- Content (documents and schema)
- Physical data structure
- Basic operations

💡 Tip

Want to get started right away? See [Create a search index](#).

Schema of a search index

In Azure AI Search, indexes contain *search documents*. Conceptually, a document is a single unit of searchable data in your index. For example, a retailer might have a document for each product, a university might have a document for each class, a travel site might have a document for each hotel and destination, and so forth. Mapping these concepts to more familiar database equivalents: a *search index* equates to a *table*, and *documents* are roughly equivalent to *rows* in a table.

The structure of a document is determined by the *index schema*, as illustrated in the following example. The "fields" collection is typically the largest part of an index, where each field is named, assigned a [data type](#), and attributed with allowable behaviors that determine how it's used.

JSON

```
{  
  "name": "name_of_index, unique across the service",  
  "description" : "Health plan coverage for standard and premium plans for Northwind  
  and Contoso employees."  
  "fields": [  
    {  
      "name": "name_of_field",  
      "type": "string",  
      "filterable": true,  
      "selectable": true,  
      " sortable": true,  
      "analyzer": "standard",  
      "normalizer": "lowercase"  
    }  
  ]  
}
```

```

    "type": "Edm.String | Collection(Edm.String) | Collection(Edm.Single) | Edm.Int32 | Edm.Int64 | Edm.Double | Edm.Boolean | Edm.DateTimeOffset | Edm.GeographyPoint",
        "searchable": true (default where applicable) | false (only Edm.String and Collection(Edm.String) fields can be searchable),
        "filterable": true (default) | false,
        "sortable": true (default where applicable) | false (Collection(Edm.String) fields cannot be sortable),
        "facetable": true (default where applicable) | false (Edm.GeographyPoint fields cannot be facetable),
        "key": true | false (default, only Edm.String fields can be keys),
        "retrievable": true (default) | false,
        "analyzer": "name_of_analyzer_for_search_and_indexing", (only if 'searchAnalyzer' and 'indexAnalyzer' are not set)
            "searchAnalyzer": "name_of_search_analyzer", (only if 'indexAnalyzer' is set and 'analyzer' is not set)
            "indexAnalyzer": "name_of_indexing_analyzer", (only if 'searchAnalyzer' is set and 'analyzer' is not set)
            "normalizer": "name_of_normalizer", (applies to fields that are filterable)
            "synonymMaps": "name_of_synonym_map", (optional, only one synonym map per field is currently supported)
        "dimensions": "number of dimensions used by an embedding models", (applies to vector fields only, of type Collection(Edm.Single))
        "vectorSearchProfile": "name_of_vector_profile" (indexes can have many configurations, a field can use just one)
    }
],
"suggesters": [ ],
"scoringProfiles": [ ],
"analyzers":(optional)[ ... ],
"charFilters":(optional)[ ... ],
"tokenizers":(optional)[ ... ],
"tokenFilters":(optional)[ ... ],
"defaultScoringProfile": (optional) "...",
"corsOptions": (optional) { },
"encryptionKey":(optional){ },
"semantic":(optional){ },
"vectorSearch":(optional){ }
}

```

Other elements are collapsed for brevity, but the following links provide details:

- [suggesters](#) support type-ahead queries like autocomplete.
- [scoringProfiles](#) are used for relevance tuning.
- [analyzers](#) are used to process strings into tokens according to linguistic rules or other characteristics supported by the analyzer.
- [corsOptions](#), or Cross-origin remote scripting (CORS), is used for apps that issues requests from different domains.
- [encryptionKey](#) configures double-encryption of sensitive content in the index.
- [semantic](#) configures semantic reranking in full text and hybrid search.
- [vectorSearch](#) configures vector fields and queries.

Field definitions

A search document is defined by the "fields" collection in the body of [Create Index request](#). You need fields for document identification (keys), storing searchable text, and fields for supporting filters, facets, and sorting. You might also need fields for data that a user never sees. For example, you might want fields for profit margins or marketing promotions that you can use in a scoring profile to boost a search score.

If incoming data is hierarchical in nature, you can represent it within an index as a [complex type](#), used for nested structures. The sample data set, [Hotels](#), illustrates complex types using an Address (contains multiple subfields) that has a one-to-one relationship with each hotel, and a Rooms complex collection, where multiple rooms are associated with each hotel.

Field attributes

Field attributes determine how a field is used, such as whether it's used in full text search, faceted navigation, sort operations, and so forth.

String fields are often marked as "searchable" and " retrievable". Fields used to narrow search results include "sortable", "filterable", and "facetable".

[\[+\] Expand table](#)

Attribute	Description
"searchable"	Full-text or vector searchable. Text fields are subject to lexical analysis such as word-breaking during indexing. If you set a searchable field to a value like "sunny day", internally it's split into the individual tokens "sunny" and "day". For details, see How full text search works .
"filterable"	Referenced in \$filter queries. Filterable fields of type <code>Edm.String</code> or <code>Collection(Edm.String)</code> don't undergo word-breaking, so comparisons are for exact matches only. For example, if you set such a field f to "sunny day", <code>\$filter=f eq 'sunny'</code> finds no matches, but <code>\$filter=f eq 'sunny day'</code> will.
"sortable"	By default the system sorts results by score, but you can configure sort based on fields in the documents. Fields of type <code>Collection(Edm.String)</code> can't be "sortable".
"facetable"	Typically used in a presentation of search results that includes a hit count by category (for example, hotels in a specific city). This option can't be used with fields of type <code>Edm.GeographyPoint</code> . Fields of type <code>Edm.String</code> that are filterable, "sortable", or "facetable" can be at most 32 kilobytes in length. For details, see Create Index (REST API) .
"key"	Unique identifier for documents within the index. Exactly one field must be chosen as the key field and it must be of type <code>Edm.String</code> .

Attribute	Description
"retrievable"	Determines whether the field can be returned in a search result. This is useful when you want to use a field (such as <i>profit margin</i>) as a filter, sorting, or scoring mechanism, but don't want the field to be visible to the end user. This attribute must be <code>true</code> for key fields.

Although you can add new fields at any time, existing field definitions are locked in for the lifetime of the index. For this reason, developers typically use the Azure portal for creating simple indexes, testing ideas, or using the Azure portal pages to look up a setting. Frequent iteration over an index design is more efficient if you follow a code-based approach so that you can rebuild the index easily.

(!) Note

The APIs you use to build an index have varying default behaviors. For the [REST APIs](#), most attributes are enabled by default (for example, "searchable" and "retrievable" are true for string fields) and you often only need to set them if you want to turn them off. For the .NET SDK, the opposite is true. On any property you do not explicitly set, the default is to disable the corresponding search behavior unless you specifically enable it.

Physical structure and size

In Azure AI Search, the physical structure of an index is largely an internal implementation. You can access its schema, load and query its content, monitor its size, and manage its capacity. However, Microsoft manages the infrastructure and physical data structures stored with your search service.

You can monitor index size on the [Search management > Indexes](#) page in the Azure portal. Alternatively, you can issue a [GET INDEX request](#) against your search service or a [Service Statistics request](#) to check the value of storage size.

The size of an index is determined by the:

- Quantity and composition of your documents.
- Attributes on individual fields: "retrievable" doesn't bloat your index, but "filterable", "sortable", "facetable" consume more storage for storing non-tokenized text.
- Index configuration. Specifically, whether you include suggesters or specialized [analyzers](#). If you use the edgeNgram tokenizer to store verbatim sequences of characters (`a`, `ab`, `abc`, `abcd`), the index is larger than if you use the standard analyzer.

Document composition and quantity are determined by what you choose to import.

Remember that a search index should only contain content that's useful for your search application. If source data includes binary fields, omit those fields unless you're using AI enrichment to crack and analyze the content to create text-searchable information.

Field attributes determine behaviors. To support those behaviors, the indexing process creates the necessary data structures. For example, for a field of type `Edm.String`, "searchable" invokes [full-text search](#), which scans inverted indexes for the tokenized term. In contrast, a "filterable" or "sortable" attribute supports iteration over unmodified strings.

Suggesters are constructs that support type-ahead or autocomplete queries. When you include a suggester, the indexing process creates the data structures necessary for verbatim character matches. Suggesters are implemented at the field level, so choose only those fields that are reasonable for type-ahead.

Basic operations and interaction

Now that you have a better idea of what an index is, this section introduces index runtime operations, including connecting to and securing a single index.

Note

There's no portal or API support for moving or copying an index. Typically, you either point your application deployment to a different search service (using the same index name) or revise the name to create a copy on your current search service and then build it.

Index isolation

In Azure AI Search, you work with one index at a time. All index-related operations target a single index. There's no concept of related indexes or the joining of independent indexes for either indexing or querying.

Continuously available

An index is immediately available for queries as soon as the first document is indexed, but it's not fully operational until all documents are indexed. Internally, an index is [distributed across partitions and executes on replicas](#). The physical index is managed internally. You manage the logical index.

An index is continuously available and can't be paused or taken offline. Because it's designed for continuous operation, updates to its content and additions to the index itself happen in real time. If a request coincides with a document update, queries might temporarily return incomplete results.

Query continuity exists for document operations, such as refreshing or deleting, and for modifications that don't affect the existing structure or integrity of an index, such as adding new fields. Structural updates, such as changing existing fields, are typically managed using a drop-and-rebuild workflow in a development environment or by creating a new version of the index on the production service.

To avoid an [index rebuild](#), some customers who are making small changes "version" a field by creating a new one that coexists with a previous version. Over time, this leads to orphaned content by way of obsolete fields and obsolete custom analyzer definitions, especially in a production index that's expensive to replicate. You can address these issues during planned updates to the index as part of index lifecycle management.

Endpoint connection and security

All indexing and query requests target an index. Endpoints are usually one of the following:

[] [Expand table](#)

Endpoint	Connection and access control
<code><your-service>.search.windows.net/indexes</code>	Targets the indexes collection. Used when creating, listing, or deleting an index. Admin rights are required for these operations and available through admin API keys or a Search Contributor role .
<code><your-service>.search.windows.net/indexes/<your-index>/docs</code>	Targets the documents collection of a single index. Used when querying an index or data refresh. For queries, read rights are sufficient and available through query API keys or a data reader role. For data refresh, admin rights are required.

How to connect to Azure AI Search

1. [Start with the Azure portal](#). Azure subscribers, or the person who created the search service, can manage the search service in the Azure portal. An Azure subscription requires Contributor or above permissions to create or delete services. This permission level is sufficient for fully managing a search service in the Azure portal.
2. Try other clients for programmatic access. We recommend the quickstarts for first steps:

- [Quickstart: REST](#)
- [Quickstart: Full-text search](#)
- [Quickstart: RAG \(using Visual Studio Code and a Jupyter notebook\)](#)

Next steps

You can get hands-on experience creating an index using almost any sample or walkthrough for Azure AI Search. For starters, you could choose any of the quickstarts from the table of contents.

But you'll also want to become familiar with methodologies for loading an index with data. Index definition and data import strategies are defined in tandem. The following articles provide more information about creating and loading an index.

- [Create a search index](#)
- [Update an index](#)
- [Create a vector store](#)
- [Create an index alias](#)
- [Data import overview](#)
- [Load an index](#)

Last updated on 12/05/2025

Vector indexes in Azure AI Search

08/28/2025

Vectors are high-dimensional embeddings that represent text, images, and other content mathematically. Azure AI Search stores vectors at the field level, allowing vector and nonvector content to coexist within the same [search index](#).

A search index becomes a *vector index* when you define vector fields and a vector configuration. To populate vector fields, you can push [precomputed embeddings](#) into them or use [integrated vectorization](#), a built-in Azure AI Search capability that generates embeddings during indexing.

At query time, the vector fields in your index enable similarity search, where the system retrieves documents whose vectors are most similar to the vector query. You can use [vector search](#) for similarity matching alone or [hybrid search](#) for a combination of similarity and keyword matching.

This article covers the key concepts for creating and managing a vector index, including:

- Vector retrieval patterns
- Content (vector fields and configuration)
- Physical data structure
- Basic operations

💡 Tip

Want to get started right away? See [Create a vector index](#).

Vector retrieval patterns

Azure AI Search supports two patterns for vector retrieval:

- **Classic search.** This pattern uses a search bar, query input, and rendered results. During query execution, the search engine or your application code vectorizes the user input. The search engine then performs vector search over the vector fields in your index and formulates a response that you render in a client app.

In Azure AI Search, results are returned as a flattened row set, and you can choose which fields to include in the response. Although the search engine matches on vectors, your index should have nonvector, human-readable content to populate the search results.

Classic search supports both [vector queries](#) and [hybrid queries](#).

- **Generative search.** Language models use data from Azure AI Search to respond to user queries. An orchestration layer typically coordinates prompts and maintains context, feeding search results into chat models like GPT. This pattern is based on the [retrieval-augmented generation \(RAG\)](#) architecture, where the search index provides grounding data.

Schema of a vector index

The schema of a vector index requires the following:

- Name
- Key field (string)
- One or more vector fields
- Vector configuration

Nonvector fields aren't required, but we recommend including them for hybrid queries or for returning verbatim content that doesn't go through a language model. For more information, see [Create a vector index](#).

Your index schema should reflect your [vector retrieval pattern](#). This section mostly covers field composition for classic search, but it also provides schema guidance for generative search.

Basic vector field configuration

Vector fields have unique data types and properties. Here's what a vector field looks like in a `fields` collection:

```
JSON

{
  "name": "content_vector",
  "type": "Collection(Edm.Single)",
  "searchable": true,
  "retrievable": true,
  "dimensions": 1536,
  "vectorSearchProfile": "my-vector-profile"
}
```

Only [certain data types](#) are supported for vector fields. The most common type is `Collection(Edm.Single)`, but using narrow types can save on storage.

Vector fields must be searchable and retrievable, but they can't be filterable, facetable, or sortable. They also can't have analyzers, normalizers, or synonym map assignments.

The `dimensions` property must be set to the number of embeddings generated by the embedding model. For example, text-embedding-ada-002 generates 1,536 embeddings for each chunk of text.

Vector fields are indexed using algorithms specified in a *vector search profile*, which is defined elsewhere in the index and not shown in this example. For more information, see [Add a vector search configuration](#).

Fields collection for basic vector workloads

Vector indexes require more than just vector fields. For example, all indexes must have a key field, which is `id` in the following example:

```
JSON

{
  "name": "example-basic-vector-idx",
  "fields": [
    { "name": "id", "type": "Edm.String", "searchable": false, "filterable": true,
      "retrievable": true, "key": true },
    { "name": "content_vector", "type": "Collection(Edm.Single)", "searchable": true,
      "retrievable": true, "dimensions": 1536, "vectorSearchProfile": null },
    { "name": "content", "type": "Edm.String", "searchable": true, "retrievable": true,
      "analyzer": null },
    { "name": "metadata", "type": "Edm.String", "searchable": true, "filterable": true,
      "retrievable": true, "sortable": true, "facetable": true }
  ]
}
```

Other fields, such as the `content` field, provide the human-readable equivalent of the `content_vector` field. If you're using language models exclusively for response formulation, you can omit nonvector content fields, but solutions that push search results directly to client apps should have nonvector content.

Metadata fields are useful for filters, especially if they include origin information about the source document. Although you can't filter directly on a vector field, you can set prefilter, postfilter, or strict postfilter (preview) modes to filter before or after vector query execution.

Schema generated by the import wizard

We recommend the [Import data \(new\) wizard](#) for evaluation and proof-of-concept testing. The wizard generates the example schema in this section.

The wizard chunks your content into smaller search documents, which benefits RAG apps that use language models to formulate responses. Chunking helps you stay within the input limits of language models and the token limits of semantic ranker. It also improves precision in

similarity search by matching queries against chunks pulled from multiple parent documents. For more information, see [Chunk large documents for vector search solutions](#).

For each search document in the following example, there's one chunk ID, parent ID, chunk, title, and vector field. The wizard:

- Populates the `chunk_id` and `parent_id` fields with base64-encoded blob metadata (path).
- Extracts the `chunk` and `title` fields from the blob content and blob name, respectively.
- Creates the `vector` field by calling an Azure OpenAI embedding model that you provide to vectorize the `chunk` field. Only the vector field is fully generated during this process.

JSON

```
"name": "example-index-from-import-wizard",
"fields": [
    { "name": "chunk_id", "type": "Edm.String", "key": true, "searchable": true,
    "filterable": true, "retrievable": true, "sortable": true, "facetable": true,
    "analyzer": "keyword"},

    { "name": "parent_id", "type": "Edm.String", "searchable": true, "filterable": true,
    "retrievable": true, "sortable": true},

    { "name": "chunk", "type": "Edm.String", "searchable": true, "filterable": false,
    "retrievable": true, "sortable": false},

    { "name": "title", "type": "Edm.String", "searchable": true, "filterable": true,
    "retrievable": true, "sortable": false},

    { "name": "vector", "type": "Collection(Edm.Single)", "searchable": true,
    "retrievable": true, "dimensions": 1536, "vectorSearchProfile": "vector-
    1707768500058-profile"}]
```

Schema for generative search

If you're designing vector storage for RAG and chat-style apps, you can create two indexes:

- One for static content that you indexed and vectorized.
- One for conversations that can be used in prompt flows.

For illustrative purposes, this section uses the [chat-with-your-data-solution-accelerator](#) to create the `chat-index` and `conversations` indexes.

Home > contosochat-search

contosochat-search | Indexes

Search service

Search management

Indexes

Add index Refresh Delete

Filter by name...

Name	Document Count	Storage Size
conversations	14	434.61 KB
chat-index	191	5.42 MB

The following fields from `chat-index` support generative search experiences:

JSON

```
"name": "example-index-from-accelerator",
"fields": [
    { "name": "id", "type": "Edm.String", "searchable": false, "filterable": true,
" retrievable": true },
    { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false,
" retrievable": true },
    { "name": "content_vector", "type": "Collection(Edm.Single)", "searchable": true,
" retrievable": true, "dimensions": 1536, "vectorSearchProfile": "my-vector-
profile" },
    { "name": "metadata", "type": "Edm.String", "searchable": true, "filterable": false,
" retrievable": true },
    { "name": "title", "type": "Edm.String", "searchable": true, "filterable": true,
" retrievable": true, "facetable": true },
    { "name": "source", "type": "Edm.String", "searchable": true, "filterable": true,
" retrievable": true },
    { "name": "chunk", "type": "Edm.Int32", "searchable": false, "filterable": true,
" retrievable": true },
    { "name": "offset", "type": "Edm.Int32", "searchable": false, "filterable": true,
" retrievable": true }
]
```

The following fields from `conversations` support orchestration and chat history:

JSON

```
"fields": [
    { "name": "id", "type": "Edm.String", "key": true, "searchable": false,
"filterable": true, "retrievable": true, "sortable": false, "facetable": false },
    { "name": "conversation_id", "type": "Edm.String", "searchable": false,
"filterable": true, "retrievable": true, "sortable": false, "facetable": true },
    { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false,
"retrievable": true },
    { "name": "content_vector", "type": "Collection(Edm.Single)", "searchable": true,
"retrievable": true, "dimensions": 1536, "vectorSearchProfile": "default-
profile" },
```

```

        {
            "name": "metadata", "type": "Edm.String", "searchable": true, "filterable": false,
            "retrievable": true },
        {
            "name": "type", "type": "Edm.String", "searchable": false, "filterable": true,
            "retrievable": true, "sortable": false, "facetable": true },
        {
            "name": "user_id", "type": "Edm.String", "searchable": false, "filterable": true,
            "retrievable": true, "sortable": false, "facetable": true },
        {
            "name": "sources", "type": "Collection(Edm.String)", "searchable": false,
            "filterable": true, "retrievable": true, "sortable": false, "facetable": true },
        {
            "name": "created_at", "type": "Edm.DateTimeOffset", "searchable": false,
            "filterable": true, "retrievable": true },
        {
            "name": "updated_at", "type": "Edm.DateTimeOffset", "searchable": false,
            "filterable": true, "retrievable": true }
    ]
]

```

The following screenshot shows search results for `conversations` in [Search explorer](#):

```

Results
18
19     {
20         "@search.score": 1,
21         "id": "ND1mODY4MjgtOWU2ZC00YTY3LTgwNTItNmI20WUyYzllZjRj",
22         "conversation_id": "01db26eb-f781-462b-8da3-0ec10e551a35",
23         "content": "The Gulf Stream carries a lot of heat from the Equator toward the far North Atlantic, m",
24         "metadata": "{\"conversation_id\": \"01db26eb-f781-462b-8da3-0ec10e551a35\", \"sources\": [\"doc_26...\"],
25         \"type\": \"assistant\",
26         \"user_id\": null,
27         \"sources\": [
28             \"doc_2005da260b463009d5e230b09a55acedf51fbcd7\",
29             \"doc_541c34b9a54b97ac888034a21c73fc6910243741\"
30         ],
31         \"created_at\": \"2024-01-15T05:19:52Z\",
32         \"updated_at\": \"2024-01-15T05:19:52Z\"
33     },
34     {
35         "@search.score": 1,
36         "id": "NDM1NWM0MzTzGzmZi00ZmQ4Ltk3YTgtY2MyMGU3YmrhNTVm",
37         "conversation_id": "01db26eb-f781-462b-8da3-0ec10e551a35",
38         "content": "what can you tell me about winds",
39         "metadata": "{\"type\": \"user\", \"conversation_id\": \"01db26eb-f781-462b-8da3-0ec10e551a35\", \"s...\"},
40         \"type\": \"user\",
41         \"user_id\": null,
42         \"sources\": [],
43         \"created_at\": \"2024-01-15T19:51:12Z\",
44         \"updated_at\": \"2024-01-15T19:51:12Z\"
45     }

```

In our example, the search score is 1.00 because the search is unqualified. Several fields support orchestration and prompt flows:

- `conversation_id` identifies each chat session.
- `type` indicates whether the content is from the user or the assistant.
- `created_at` and `updated_at` age out chats from the history.

Physical structure and size

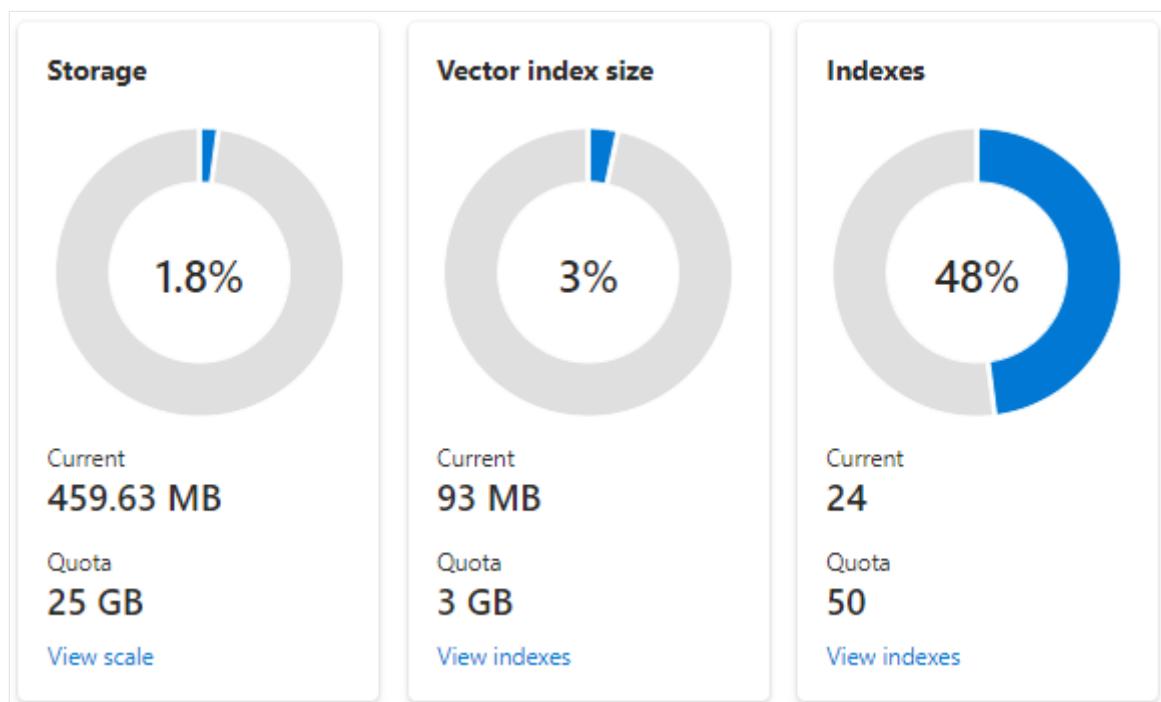
In Azure AI Search, the physical structure of an index is largely an internal implementation. You can access its schema, load and query its content, monitor its size, and manage its capacity. However, Microsoft manages the infrastructure and physical data structures stored with your search service.

The size and substance of an index are determined by the:

- Quantity and composition of your documents.
- Attributes on individual fields. For example, more storage is required for filterable fields.
- Index configuration, including the vector configuration that specifies how the internal navigation structures are created. You can choose HNSW or exhaustive KNN for similarity search.

Azure AI Search imposes limits on vector storage, which helps maintain a balanced and stable system for all workloads. To help you stay under the limits, vector usage is tracked and reported separately in the Azure portal and programmatically through service and index statistics.

The following screenshot shows an S1 service configured with one partition and one replica. This service has 24 small indexes, each with an average of one vector field consisting of 1,536 embeddings. The second tile shows the quota and usage for vector indexes. Because a vector index is an internal data structure created for each vector field, storage for vector indexes is always a fraction of the overall storage used by the index. Nonvector fields and other data structures consume the rest.



Vector index limits and estimations are covered in [another article](#), but two points to emphasize are that maximum storage depends on the creation date and pricing tier of your search service. Newer same-tier services have significantly more capacity for vector indexes. For these reasons, you should:

- [Check the creation date of your search service](#). If it was created before April 3, 2024, you might be able to [upgrade your service](#) for greater capacity.

- Choose a scalable tier if you anticipate fluctuations in vector storage requirements. For older search services, the Basic tier is fixed at one partition. Consider Standard 1 (S1) and higher for more flexibility and faster performance. You can also [switch between Basic and Standard \(S1, S2, and S3\) tiers](#).

Basic operations and interaction

This section introduces vector runtime operations, including connecting to and securing a single index.

ⓘ Note

There's no portal or API support for moving or copying an index. Typically, you either point your application deployment to a different search service (using the same index name) or revise the name to create a copy on your current search service and then build it.

Index isolation

In Azure AI Search, you work with one index at a time. All index-related operations target a single index. There's no concept of related indexes or the joining of independent indexes for either indexing or querying.

Continuously available

An index is immediately available for queries as soon as the first document is indexed, but it's not fully operational until all documents are indexed. Internally, an index is [distributed across partitions and executes on replicas](#). The physical index is managed internally. You manage the logical index.

An index is continuously available and can't be paused or taken offline. Because it's designed for continuous operation, updates to its content and additions to the index itself happen in real time. If a request coincides with a document update, queries might temporarily return incomplete results.

Query continuity exists for document operations, such as refreshing or deleting, and for modifications that don't affect the existing structure or integrity of an index, such as adding new fields. Structural updates, such as changing existing fields, are typically managed using a drop-and-rebuild workflow in a development environment or by creating a new version of the index on the production service.

To avoid an [index rebuild](#), some customers who are making small changes "version" a field by creating a new one that coexists with a previous version. Over time, this leads to orphaned content by way of obsolete fields and obsolete custom analyzer definitions, especially in a production index that's expensive to replicate. You can address these issues during planned updates to the index as part of index lifecycle management.

Endpoint connection

All vector indexing and query requests target an index. Endpoints are usually one of the following:

 [Expand table](#)

Endpoint	Connection and access control
<code><your-service>.search.windows.net/indexes</code>	Targets the indexes collection. Used when creating, listing, or deleting an index. Admin rights are required for these operations and available through admin API keys or a Search Contributor role .
<code><your-service>.search.windows.net/indexes/<your-index>/docs</code>	Targets the documents collection of a single index. Used when querying an index or data refresh. For queries, read rights are sufficient and available through query API keys or a data reader role. For data refresh, admin rights are required.

How to connect to Azure AI Search

1. [Make sure you have permissions](#) or an [API access key](#). Unless you're querying an existing index, you need admin rights or a Contributor role assignment to manage and view content on a search service.
2. [Start with the Azure portal](#). The person who created the search service can view and manage it, including granting access to others on the [Access control \(IAM\)](#) page.
3. Move on to other clients for programmatic access. For first steps, we recommend [Quickstart: Vector searching using REST](#) and the [azure-search-vector-samples](#) repo.

Manage vector stores

Azure provides a [monitoring platform](#) that includes diagnostic logging and alerting. We recommend that you:

- [Enable diagnostic logging](#).

- Set up alerts.
- Analyze query and index performance.

Secure access to vector data

Azure AI Search implements data encryption, private connections for no-internet scenarios, and role assignments for secure access through Microsoft Entra ID. For more information about enterprise security features, see [Security in Azure AI Search](#).

Related content

- [Quickstart: Vector search using REST](#)
- [Create a vector index](#)
- [Query a vector index](#)

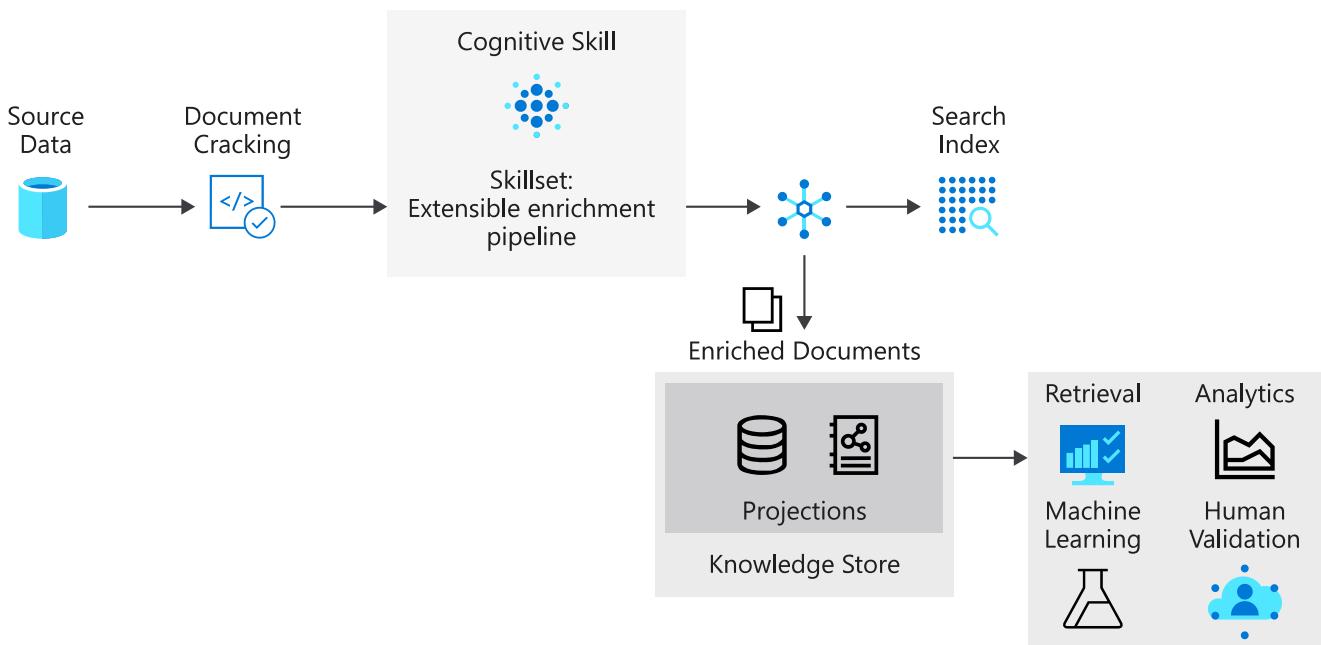
Knowledge store in Azure AI Search

! Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in agentic retrieval workflows.

Knowledge store is secondary storage for [AI-enriched content created by a skillset](#) in Azure AI Search. In Azure AI Search, an indexing job always sends output to a search index, but if you attach a skillset to an indexer, you can optionally also send AI-enriched output to a container or table in Azure Storage. A knowledge store can be used for independent analysis or downstream processing in non-search scenarios like knowledge mining.

The two outputs of indexing, a search index and knowledge store, are mutually exclusive products of the same pipeline. They're derived from the same inputs and contain the same data, but their content is structured, stored, and used in different applications.



Physically, a knowledge store is [Azure Storage](#), either Azure Table Storage, Azure Blob Storage, or both. Any tool or process that can connect to Azure Storage can consume the contents of a knowledge store. There's no query support in Azure AI Search for retrieving content from a knowledge store.

When viewed through Azure portal, a knowledge store looks like any other collection of tables, objects, or files. The following screenshot shows a knowledge store composed of three tables. You can adopt a naming convention, such as a `kstore` prefix, to keep your content together.

The screenshot shows the Azure Storage browser (preview) interface for a storage account named 'demoblobstorage'. The left sidebar includes links for Overview, Activity log, Tags, Diagnose and solve problems, Access Control (IAM), Data migration, Events, and Storage browser (preview). The 'Storage browser (preview)' link is highlighted with a red box. The main area displays a list of storage entities under 'Tables'. A red box highlights the 'Tables' link in the navigation bar and the list of tables. The tables listed are: kstoreProjectionDemoDocument, kstoreProjectionDemoEntities, kstoreProjectionDemoKeyPhrases, and MsAzSearchIndexerCacheIndex33b0d... .

Benefits of knowledge store

The primary benefits of a knowledge store are two-fold: flexible access to content, and the ability to shape data.

Unlike a search index that can only be accessed through queries in Azure AI Search, a knowledge store is accessible to any tool, app, or process that supports connections to Azure Storage. This flexibility opens up new scenarios for consuming the analyzed and enriched content produced by an enrichment pipeline.

The same skillset that enriches data can also be used to shape data. Some tools like Power BI work better with tables, whereas a data science workload might require a complex data structure in a blob format. Adding a **Shaper skill** to a skillset gives you control over the shape of your data. You can then pass these shapes to projections, either tables or blobs, to create physical data structures that align with the data's intended use.

The following video explains both of these benefits and more.

<https://www.youtube-nocookie.com/embed/XWzLBP8iWqg?version=3>

Knowledge store definition

A knowledge store is defined inside a skillset definition and it has two components:

- A connection string to Azure Storage

- **Projections** that determine whether the knowledge store consists of tables, objects or files. The projections element is an array. You can create multiple sets of table-object-file combinations within one knowledge store.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "<YOUR-AZURE-STORAGE-ACCOUNT-CONNECTION-STRING>",
  "projections": [
    {
      "tables": [ ],
      "objects": [ ],
      "files": [ ]
    }
  ]
}
```

The type of projection you specify in this structure determines the type of storage used by knowledge store, but not its structure. Fields in tables, objects, and files are determined by Shaper skill output if you're creating the knowledge store programmatically, or by the Import data wizard if you're using the Azure portal.

- `tables` project enriched content into Table Storage. Define a table projection when you need tabular reporting structures for inputs to analytical tools or export as data frames to other data stores. You can specify multiple `tables` within the same projection group to get a subset or cross section of enriched documents. Within the same projection group, table relationships are preserved so that you can work with all of them.

Projected content isn't aggregated or normalized. The following screenshot shows a table, sorted by key phrase, with the parent document indicated in the adjacent column. In contrast with data ingestion during indexing, there's no linguistic analysis or aggregation of content. Plural forms and differences in casing are considered unique instances.

Content.metadata_storage_name	Content.KeyPhrases
Cognitive Services and Content Intelligence.pptx	Computer Vision
10-K-FY16.html	computing device
10-K-FY16.html	computing devices
MSFT_FY17_10K.docx	computing devices
10-K-FY16.html	Computing segment
Cognitive Services and Bots (spanish).pdf	confianza

- `objects` project JSON document into Blob storage. The physical representation of an `object` is a hierarchical JSON structure that represents an enriched document.
- `files` project image files into Blob storage. A `file` is an image extracted from a document, transferred intact to Blob storage. Although it's named "files", it shows up in Blob Storage, not file storage.

Create a knowledge store

To create knowledge store, use the Azure portal or an API.

You need [Azure Storage](#), a [skillset](#), and an [indexer](#). Because indexers require a search index, you also need to provide an [index definition](#).

Go with the Azure portal approach for the fastest route to a finished knowledge store. Or, choose the REST API for a deeper understanding of how objects are defined and related.

Azure portal

[Create your first knowledge store in four steps](#) using the [Import data wizard](#).

1. Define a data source that contains the data you want to enrich.
2. Define a skillset. The skillset specifies enrichment steps and the knowledge store.
3. Define an index schema. You might not need one, but indexers require it. The wizard can infer an index.
4. Complete the wizard. Data extraction, enrichment, and knowledge store creation occur in this last step.

The wizard automates several tasks. Specifically, both data shaping and projections (definitions of physical data structures in Azure Storage) are created for you.

Connect with apps

Once enriched content exists in storage, any tool or technology that connects to Azure Storage can be used to explore, analyze, or consume the contents. The following list is a start:

- [Storage Explorer](#) or Storage browser in the Azure portal to view enriched document structure and content. Consider this as your baseline tool for viewing knowledge store contents.

- [Power BI](#) for reporting and analysis.
- [Azure Data Factory](#) for further manipulation.

Content lifecycle

Each time you run the indexer and skillset, the knowledge store is updated if the skillset or underlying source data has changed. Any changes picked up by the indexer are propagated through the enrichment process to the projections in the knowledge store, ensuring that your projected data is a current representation of content in the originating data source.

Note

While you can edit the data in the projections, any edits will be overwritten on the next pipeline invocation, assuming the document in source data is updated.

Changes in source data

For data sources that support change tracking, an indexer will process new and changed documents, and bypass existing documents that have already been processed. Timestamp information varies by data source, but in a blob container, the indexer looks at the `lastmodified` date to determine which blobs need to be ingested.

Changes to a skillset

If you're making changes to a skillset, you should [enable caching of enriched documents](#) to reuse existing enrichments where possible.

Without incremental caching, the indexer will always process documents in order of the high water mark, without going backwards. For blobs, the indexer would process blobs sorted by `lastModified`, regardless of any changes to indexer settings or the skillset. If you change a skillset, previously processed documents aren't updated to reflect the new skillset. Documents processed after the skillset change will use the new skillset, resulting in index documents being a mix of old and new skillsets.

With incremental caching, and after a skillset update, the indexer will reuse any enrichments that are unaffected by the skillset change. Upstream enrichments are pulled from cache, as are any enrichments that are independent and isolated from the skill that was changed.

Deletions

Although an indexer creates and updates structures and content in Azure Storage, it doesn't delete them. Projections continue to exist even when the indexer or skillset is deleted. As the owner of the storage account, you should delete a projection if it's no longer needed.

Next steps

Knowledge store offers persistence of enriched documents, useful when designing a skillset, or the creation of new structures and content for consumption by any client applications capable of accessing an Azure Storage account.

The simplest approach for creating enriched documents is [through the Azure portal](#), but a REST client and REST APIs can provide more insight into how objects are created and referenced programmatically.

[Create a knowledge store using REST](#)

Last updated on 10/21/2025

Indexers in Azure AI Search

06/23/2025

An *indexer* in Azure AI Search is a crawler that extracts textual data from cloud data sources and populates a search index using field-to-field mappings between source data and a search index. This approach is sometimes referred to as a 'pull model' because the search service pulls data in without you having to write any code that adds data to an index.

Indexers also drive [skillset execution and AI enrichment](#), where you can configure skills to integrate extra processing of content en route to an index. A few examples are OCR over image files, text split skill for data chunking, and calling embedding models to generate vectors for vector search.

Indexers target [supported data sources](#). An indexer configuration specifies a data source (origin) and a search index (destination). Several sources, such as Azure Blob Storage, have more indexer configuration properties specific to that content type.

You can run indexers on demand or on a recurring data refresh schedule that runs as often as every five minutes. More frequent updates preclude the use of indexers, requiring that you implement a '[push model](#)' that simultaneously pushes data to both Azure AI Search and your external data source for data synchronization.

A search service runs one indexer job per search unit. If you need concurrent processing, make sure you have [sufficient replicas](#). Indexers don't run in the background, so you might detect more query throttling than usual if the service is under pressure.

Indexer scenarios and use cases

You can use an indexer as the sole means for data ingestion, or in combination with other techniques. The following table summarizes the main scenarios.

[] [Expand table](#)

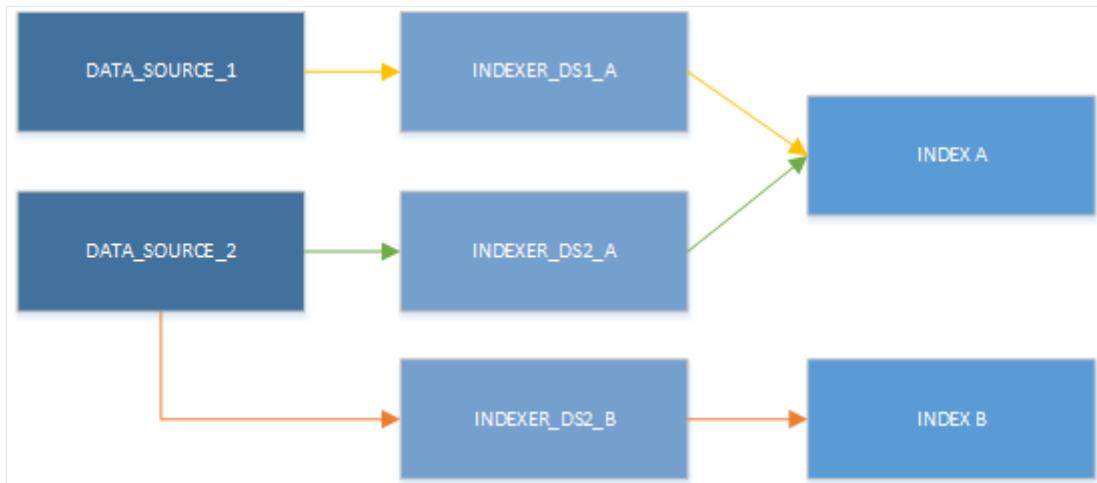
Scenario	Strategy
Single data source	This pattern is the simplest: one data source is the sole content provider for a search index. Most supported data sources provide some form of change detection so that subsequent indexer runs pick up the difference when content is added or updated in the source.
Multiple data sources	An indexer specification can have only one data source, but the search index itself can accept content from multiple sources, where each indexer job brings new content from a different data provider. Each source can contribute its share of full documents,

Scenario	Strategy
	<p>or populate selected fields in each document. For a closer look at this scenario, see Tutorial: Index from multiple data sources.</p>
Multiple indexers	<p>Multiple data sources are typically paired with multiple indexers if you need to vary run time parameters, the schedule, or field mappings.</p> <p>Cross-region scale out of Azure AI Search is a variation of this scenario. You might have copies of the same search index in different regions. To synchronize search index content, you could have multiple indexers pulling from the same data source, where each indexer targets a different search index in each region.</p> <p>Parallel indexing of very large data sets also requires a multi-indexer strategy, where each indexer targets a subset of the data.</p>
Content transformation	<p>Indexers drive skillset execution and AI enrichment. Content transforms are defined in a skillset that you attach to the indexer. You can use skills to incorporate data chunking and vectorization.</p>

You should plan on creating one indexer for every target index and data source combination. You can have multiple indexers writing into the same index, and you can reuse the same data source for multiple indexers. However, an indexer can only consume one data source at a time, and can only write to a single index. As the following graphic illustrates, one data source provides input to one indexer, which then populates a single index:



Although you can only use one indexer at a time, resources can be used in different combinations. The main takeaway of the next illustration is to notice is that a data source can be paired with more than one indexer, and multiple indexers can write to same index.



Supported data sources

Indexers crawl data stores on Azure and outside of Azure.

- [Azure Blob Storage](#)
- [Azure Cosmos DB](#)
- [Azure Data Lake Storage Gen2](#)
- [Azure SQL Database](#)
- [Azure Table Storage](#)
- [Azure SQL Managed Instance](#)
- [SQL Server on Azure Virtual Machines](#)
- [Azure Files \(in preview\)](#)
- [Azure MySQL \(in preview\)](#)
- [SharePoint in Microsoft 365 \(in preview\)](#)
- [Azure Cosmos DB for MongoDB \(in preview\)](#)
- [Azure Cosmos DB for Apache Gremlin \(in preview\)](#)
- [OneLake \(in preview\)](#)

Azure Cosmos DB for Cassandra is not supported.

Indexers accept flattened row sets, such as a table or view, or items in a container or folder. In most cases, it creates one search document per row, record, or item.

Indexer connections to remote data sources can be made using standard Internet connections (public) or encrypted private connections when you use a shared private link. You can also set up connections to authenticate using a managed identity. For more information about secure connections, see [Indexer access to content protected by Azure network security features](#) and [Connect to a data source using a managed identity](#).

Stages of indexing

On an initial run, when the index is empty, an indexer will read in all of the data provided in the table or container. On subsequent runs, the indexer can usually detect and retrieve just the data that has changed. For blob data, change detection is automatic. For other data sources like Azure SQL or Azure Cosmos DB, change detection must be enabled.

For each document it receives, an indexer implements or coordinates multiple steps, from document retrieval to a final search engine "handoff" for indexing. Optionally, an indexer also drives [skillset execution and outputs](#), assuming a skillset is defined.



Stage 1: Document cracking

Document cracking is the process of opening files and extracting content. Text-based content can be extracted from files on a service, rows in a table, or items in container or collection.

You can also enable image extraction during document cracking for an [extra fee ↗](#). This is disabled by default and can be enabled via the `imageAction` property in the [indexer parameters configuration](#). Review some [image scenarios](#) for indexer image handling.

Depending on the data source, the indexer will try different operations to extract potentially indexable content:

- When the document is a file with embedded images, such as a PDF, the indexer extracts text, images, and metadata. Indexers can open files from [Azure Blob Storage](#), [Azure Data Lake Storage Gen2](#), and [SharePoint](#).
- When the document is a record in [Azure SQL](#), the indexer will extract non-binary content from each field in each record.
- When the document is a record in [Azure Cosmos DB](#), the indexer will extract non-binary content from fields and subfields from the Azure Cosmos DB document.

Note that the document cracking process can also be triggered later during the optional [skillset execution](#) stage, using skillsets, for data transformation. Adding a skillset with [image skills](#) allows document cracking to extract images and queue them for processing.

Stage 2: Field mappings

An indexer extracts text from a source field and sends it to a destination field in an index or knowledge store. When field names and data types coincide, the path is clear. However, you might want different names or types in the output, in which case you need to tell the indexer how to map the field.

To [specify field mappings](#), enter the source and destination fields in the indexer definition.

Field mapping occurs after document cracking, but before transformations, when the indexer is reading from the source documents. When you define a field mapping, the value of the source field is sent as-is to the destination field with no modifications.

Stage 3: Skillset execution

Skillset execution is an optional step that invokes built-in or custom AI processing. Skillsets can add optical character recognition (OCR) or other forms of image analysis if the content is binary. Skillsets can also add natural language processing. For example, you can add text translation or key phrase extraction.

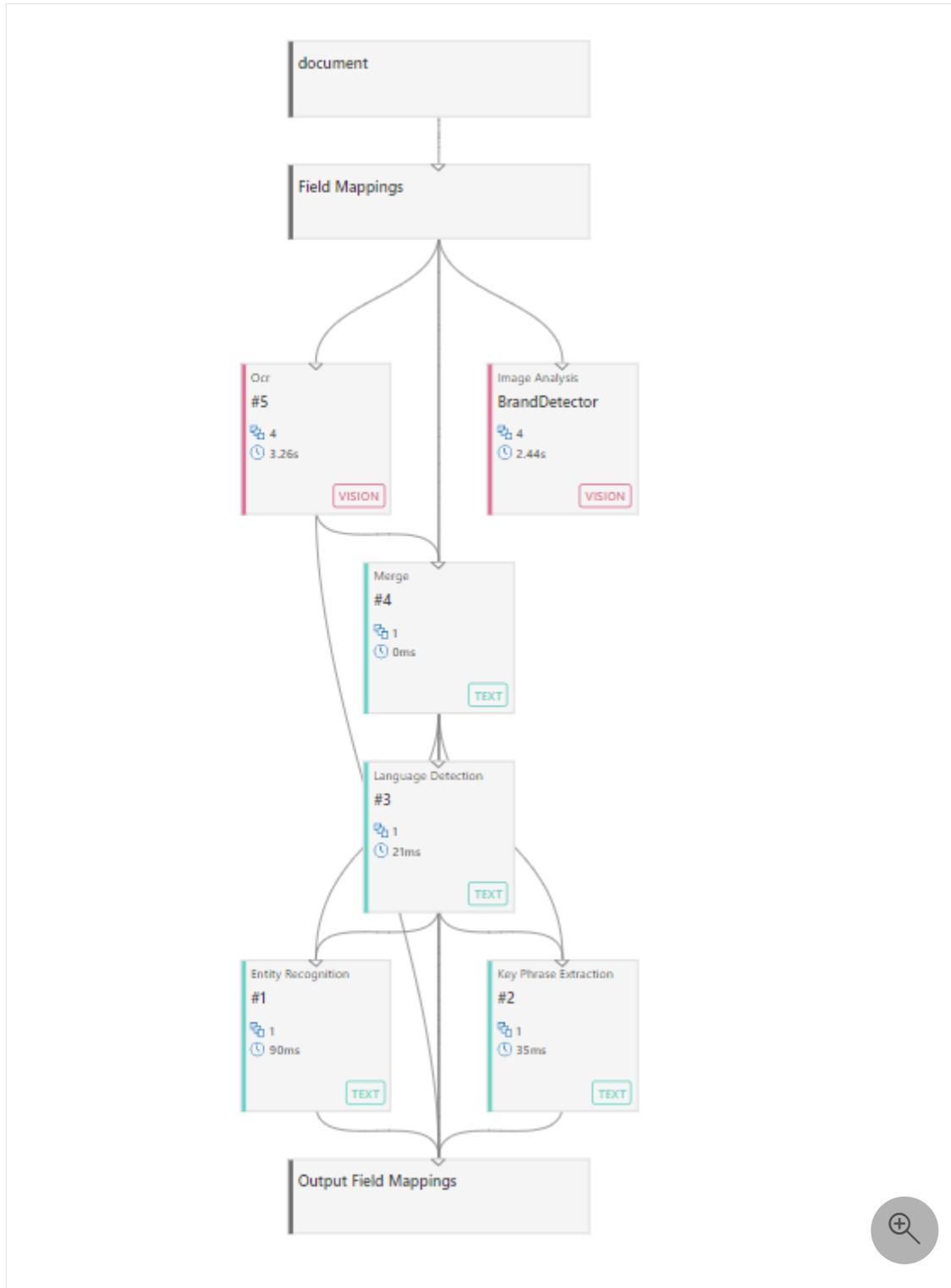
Whatever the transformation, skillset execution is where enrichment occurs. If an indexer is a pipeline, you can think of a [skillset](#) as a "pipeline within the pipeline".

Stage 4: Output field mappings

If you include a skillset, you'll need to [specify output field mappings](#) in the indexer definition. The output of a skillset is manifested internally as a tree structure referred to as an *enriched document*. Output field mappings allow you to select which parts of this tree to map into fields in your index.

Despite the similarity in names, output field mappings and field mappings build associations from different sources. Field mappings associate the content of source field to a destination field in a search index. Output field mappings associate the content of an internal enriched document (skill outputs) to destination fields in the index. Unlike field mappings, which are considered optional, an output field mapping is required for any transformed content that should be in the index.

The next image shows a sample indexer [debug session](#) representation of the indexer stages: document cracking, field mappings, skillset execution, and output field mappings.



Basic workflow

Indexers can offer features that are unique to the data source. In this respect, some aspects of indexer or data source configuration will vary by indexer type. However, all indexers share the same basic composition and requirements. Steps that are common to all indexers are covered below.

Step 1: Create a data source

Indexers require a *data source* object that provides a connection string and possibly credentials. Data sources are independent objects. Multiple indexers can use the same data source object to load more than one index at a time.

You can create a data source using any of these approaches:

- Using the Azure portal, on the **Data sources** tab of your search service pages, select **Add data source** to specify the data source definition.
- Using the Azure portal, the [Import data wizard](#) outputs a data source.
- Using the REST APIs, call [Create Data Source](#).
- Using the Azure SDK for .NET, call [SearchIndexerDataSourceConnection class](#)

Step 2: Create an index

An indexer will automate some tasks related to data ingestion, but creating an index is generally not one of them. As a prerequisite, you must have a predefined index that contains corresponding target fields for any source fields in your external data source. Fields need to match by name and data type. If not, you can [define field mappings](#) to establish the association.

For more information, see [Create an index](#).

Step 3: Create and run (or schedule) the indexer

An indexer definition consists of properties that uniquely identify the indexer, specify which data source and index to use, and provide other configuration options that influence run time behaviors, including whether the indexer runs on demand or on a schedule.

Any errors or warnings about data access or skillset validation will occur during indexer execution. Until indexer execution starts, dependent objects such as data sources, indexes, and skillsets are passive on the search service.

For more information, see [Create an indexer](#)

After the first indexer run, you can [rerun it on demand](#) or [set up a schedule](#).

You can monitor [indexer status in the Azure portal](#) or through [Get Indexer Status API](#). You should also [run queries on the index](#) to verify the result is what you expected.

Indexers don't have dedicated processing resources. Based on this, indexers' status may show as idle before running (depending on other jobs in the queue) and run times may not be predictable. Other factors define indexer performance as well, such as document size, document complexity, image analysis, among others.

Next steps

Now that you've been introduced to indexers, a next step is to review indexer properties and parameters, scheduling, and indexer monitoring. Alternatively, you could return to the list of [supported data sources](#) for more information about a specific source.

- [Create indexers](#)
- [Reset and run indexers](#)
- [Schedule indexers](#)
- [Define field mappings](#)
- [Monitor indexer status](#)

AI enrichment in Azure AI Search

In Azure AI Search, *AI enrichment* refers to integration with [Foundry Tools](#) to process content that isn't searchable in its raw form. Through enrichment, analysis and inference are used to create searchable content and structure where none previously existed.

Because Azure AI Search is used for text and vector queries, the purpose of AI enrichment is to improve the utility of your content in search-related scenarios. Raw content must be text or images (you can't enrich vectors), but the output of an enrichment pipeline can be vectorized and indexed in a search index using skills like [Text Split skill](#) for chunking and [Azure OpenAI Embedding skill](#) for vector encoding. For more information about using skills in vector scenarios, see [Integrated data chunking and embedding](#).

AI enrichment is based on [skills](#).

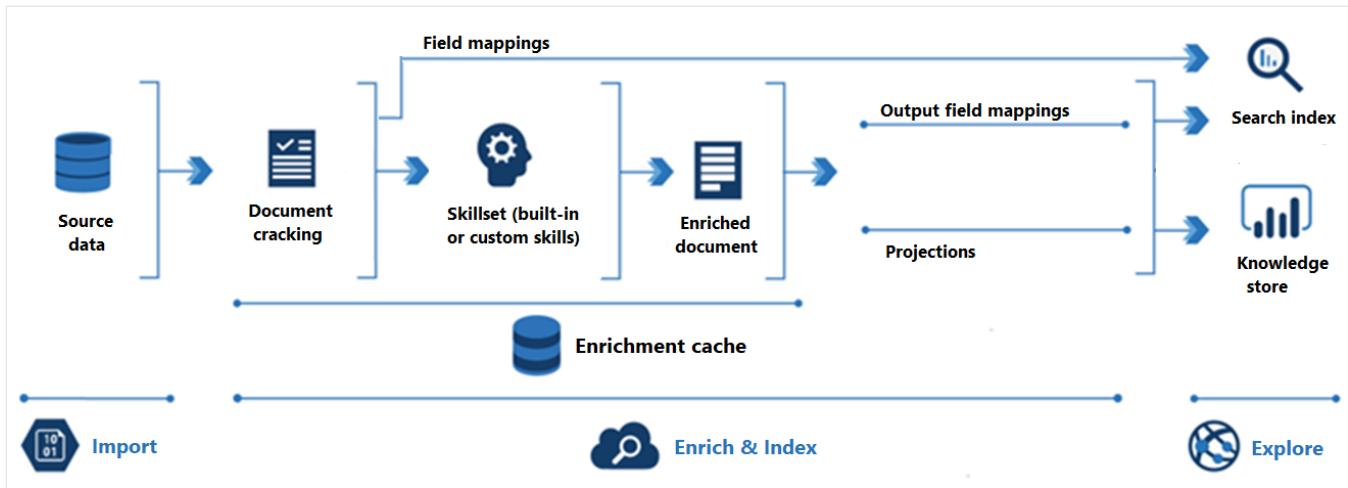
[Built-in skills](#) tap Foundry Tools. They apply the following transformations and processing to raw content:

- Translation and language detection for multilingual search.
- Entity recognition to extract people names, places, and other entities from large chunks of text.
- Key phrase extraction to identify and output important terms.
- Optical character recognition (OCR) to recognize printed and handwritten text in binary files.
- Image analysis to describe image content and output the descriptions as searchable text fields.
- Text embeddings via Azure OpenAI for integrated vectorization.
- Multimodal embeddings via Azure Vision in Foundry Tools for text and image vectorization.

Custom skills run your external code. You can use custom skills for any custom processing you want to include in the pipeline.

AI enrichment is an extension of an [indexer pipeline](#) that connects to Azure data sources. An enrichment pipeline has all of the components of an indexer pipeline (indexer, data source, index) and a [skillset](#) that specifies atomic enrichment steps.

The following diagram shows the progression of AI enrichment:



Import is the first step. Here, the indexer connects to a data source and pulls content (documents) into the search service. [Azure Blob Storage](#) is the most common resource used in AI enrichment scenarios, but any supported data source can provide content.

Enrich & Index covers most of the AI enrichment pipeline:

- Enrichment starts when the indexer [cracks documents](#) and extracts images and text. The type of processing that occurs next depends on your data and the skills you've added to a skillset. Images can be forwarded to [skills that perform image processing](#). Text content is queued for text and natural language processing. Internally, skills create an [enriched document](#) that collects transformations as they occur.
- Enriched content is generated during skillset execution and is temporary unless you save it. You can enable an [enrichment cache](#) to persist skill outputs for reuse in future skillset executions.
- To get content into a search index, the indexer must have mapping information for sending enriched content to target field. [Field mappings](#) (explicit or implicit) set the data path from source data to a search index. [Output field mappings](#) set the data path from enriched documents to an index.
- Indexing is the process wherein raw and enriched content is ingested into the physical data structures of a [search index](#) (its files and folders). Lexical analysis and tokenization occur in this step.

Exploration is the last step. Output is always a [search index](#) that you can query from a client app. Output can optionally be a [knowledge store](#) consisting of blobs and tables in Azure Storage that are accessed through data exploration tools or downstream processes. If you're creating a knowledge store, [projections](#) determine the data path for enriched content. The same enriched content can appear in both indexes and knowledge stores.

When to use AI enrichment

Enrichment is useful if raw content is unstructured text, image content, or content that needs language detection and translation. Applying AI through the [built-in skills](#) can unlock this content for full-text search and data science applications.

You can also create [custom skills](#) to provide external processing. Open-source, third-party, or first-party code can be integrated into the pipeline as a custom skill. Classification models that identify salient characteristics of various document types fall into this category, but any external package that adds value to your content could be used.

Use-cases for built-in skills

Built-in skills are based on the Foundry Tools APIs: [Azure Vision](#) and [Azure Language](#). Unless your content input is small, expect to [attach a billable Microsoft Foundry resource](#) to run larger workloads.

A [skillset](#) that's assembled using built-in skills is well suited for the following application scenarios:

- **Image processing** skills include [Optical Character Recognition \(OCR\)](#) and identification of [visual features](#), such as facial detection, image interpretation, image recognition (famous people and landmarks), or attributes like image orientation. These skills create text representations of image content for full-text search in Azure AI Search.
- **Machine translation** is provided by the [Text Translation](#) skill, often paired with [language detection](#) for multi-language solutions.
- **Natural language processing** analyzes chunks of text. Skills in this category include [Entity Recognition](#), [Sentiment Detection \(including opinion mining\)](#), and [Personal Identifiable Information Detection](#). With these skills, unstructured text is mapped as searchable and filterable fields in an index.

Use-cases for custom skills

[Custom skills](#) execute external code that you provide and wrap in the [custom skill web interface](#). Several examples of custom skills can be found in the [azure-search-power-skills](#) GitHub repository.

Custom skills aren't always complex. For example, if you have an existing package that provides pattern matching or a document classification model, you can wrap it in a custom skill.

Storing output

In Azure AI Search, an indexer saves the output it creates. A single indexer run can create up to three data structures that contain enriched and indexed output.

[+] [Expand table](#)

Data store	Required	Location	Description
searchable index	Required	Search service	Used for full-text search and other query forms. Specifying an index is an indexer requirement. Index content is populated from skill outputs, plus any source fields that are mapped directly to fields in the index.
knowledge store	Optional	Azure Storage	Used for downstream apps like knowledge mining, data science, and multimodal search. A knowledge store is defined within a skillset. Its definition determines whether your enriched documents are projected as tables or objects (files or blobs) in Azure Storage. For multimodal search scenarios , you can save extracted images to the knowledge store and reference them at query time, allowing the images to be returned directly to client apps.
enrichment cache	Optional	Azure Storage	Used for caching enrichments for reuse in subsequent skillset executions. The cache stores imported, unprocessed content (cracked documents). It also stores the enriched documents created during skillset execution. Caching is helpful if you're using image analysis or OCR, and you want to avoid the time and expense of reprocessing image files.

Indexes and knowledge stores are fully independent of each other. While you must attach an index to satisfy indexer requirements, if your sole objective is a knowledge store, you can ignore the index after it's populated.

Exploring content

After you define and load a [search index](#) or [knowledge store](#), you can explore its data.

Query a search index

[Run queries](#) to access the enriched content generated by the pipeline. The index is like any other you might create for Azure AI Search: you can supplement text analysis with custom analyzers, invoke fuzzy search queries, add filters, or experiment with scoring profiles to tune search relevance.

Use data exploration tools on a knowledge store

In Azure Storage, a [knowledge store](#) can assume the following forms: a blob container of JSON documents, a blob container of image objects, or tables in Table Storage. You can use [Storage Explorer](#), [Power BI](#), or any app that connects to Azure Storage to access your content.

- A blob container captures enriched documents in their entirety, which is useful if you're creating a feed into other processes.
- A table is useful if you need slices of enriched documents, or if you want to include or exclude specific parts of the output. For analysis in Power BI, tables are the recommended data source for data exploration and visualization in Power BI.

Availability and pricing

AI enrichment is available in regions that offer Foundry Tools. To check the availability of AI enrichment, see the [regions list](#).

Billing follows a Standard pricing model. Costs associated with built-in skills are incurred when you specify an Azure OpenAI in Foundry Models resource or Foundry resource key in the skillset. There are also costs associated with image extraction, as metered by Azure AI Search. However, text extraction and utility skills aren't billable. For more information, see [How you're charged for Azure AI Search](#).

Checklist: A typical workflow

An enrichment pipeline consists of [indexers](#) that have [skillsets](#). Post-indexing, you can query an index to validate your results.

Start with a subset of data in a [supported data source](#). Indexer and skillset design is an iterative process. The work goes faster with a small representative data set.

1. Create a [data source](#) that specifies a connection to your data.
2. [Create a skillset](#). Unless your project is small, you should [attach a Foundry resource](#). If you're [creating a knowledge store](#), define it within the skillset.
3. [Create an index schema](#) that defines a search index.
4. [Create and run the indexer](#) to bring all of the previous components together. This step retrieves the data, runs the skillset, and loads the index.

An indexer is also where you specify field mappings and output field mappings that set up the data path to a search index.

Optionally, [enable enrichment caching](#) in the indexer configuration. This step allows you to reuse existing enrichments later on.

5. [Run queries](#) to evaluate results or [start a debug session](#) to work through any skillset issues.

To repeat any of the previous steps, [reset the indexer](#) before you run it. Alternatively, you can delete and recreate the objects on each run (recommended if you're using the free tier). If you enabled caching, the indexer pulls from the cache if the source data is unchanged and if your edits to the pipeline don't invalidate the cache.

Next steps

- [Quickstart: Create a skillset for AI enrichment](#)
- [Tutorial: Learn about the AI enrichment REST APIs](#)
- [Skillset concepts](#)
- [Knowledge store concepts](#)
- [Create a skillset](#)
- [Create a knowledge store](#)

Last updated on 11/18/2025

Integrated vector embedding in Azure AI Search

Integrated vectorization is an extension of the indexing and query pipelines in Azure AI Search. It adds the following capabilities:

- Vector encoding during indexing
- Vector encoding during queries

[Data chunking](#) isn't a hard requirement, but unless your raw documents are small, chunking is necessary for meeting the token input requirements of embedding models.

Vector conversions are one-way: nonvector-to-vector. For example, there's no vector-to-text conversion for queries or results, such as converting a vector result to a human-readable string, which is why indexes contain both vector and nonvector fields.

Integrated vectorization speeds up the development and minimizes maintenance tasks during data ingestion and query time because there are fewer operations that you have to implement manually. This capability is now generally available.

Using integrated vectorization during indexing

For integrated data chunking and vector conversions, you're taking a dependency on the following components:

- An [indexer](#), which retrieves raw data from a [supported data source](#) and drives the pipeline engine.
- A [search index](#) to receive the chunked and vectorized content.
- A [skillset](#) configured for:
 - [Text Split skill](#), used to chunk the data.
 - An embedding skill, used to generate vector arrays, which can be any of the following:
 - [AzureOpenAIEmbedding skill](#), attached to text-embedding-ada-002, text-embedding-3-small, text-embedding-3-large on Azure OpenAI.
 - [Custom skill](#) that points to another embedding model on Azure or on another site.
 - [Azure Vision multimodal embeddings skill \(preview\)](#) that points to the multimodal API for Azure Vision.

- [AML skill](#) that points to select models in the Microsoft Foundry model catalog.

Using integrated vectorization in queries

For text-to-vector conversion during queries, you take a dependency on these components:

- A query that specifies one or more vector fields.
- A text string that's converted to a vector at query time.
- [A vectorizer](#), defined in the index schema, assigned to a vector field, and used automatically at query time to convert a text query to a vector. The vectorizer you set up must match the embedding model used to encode your content.

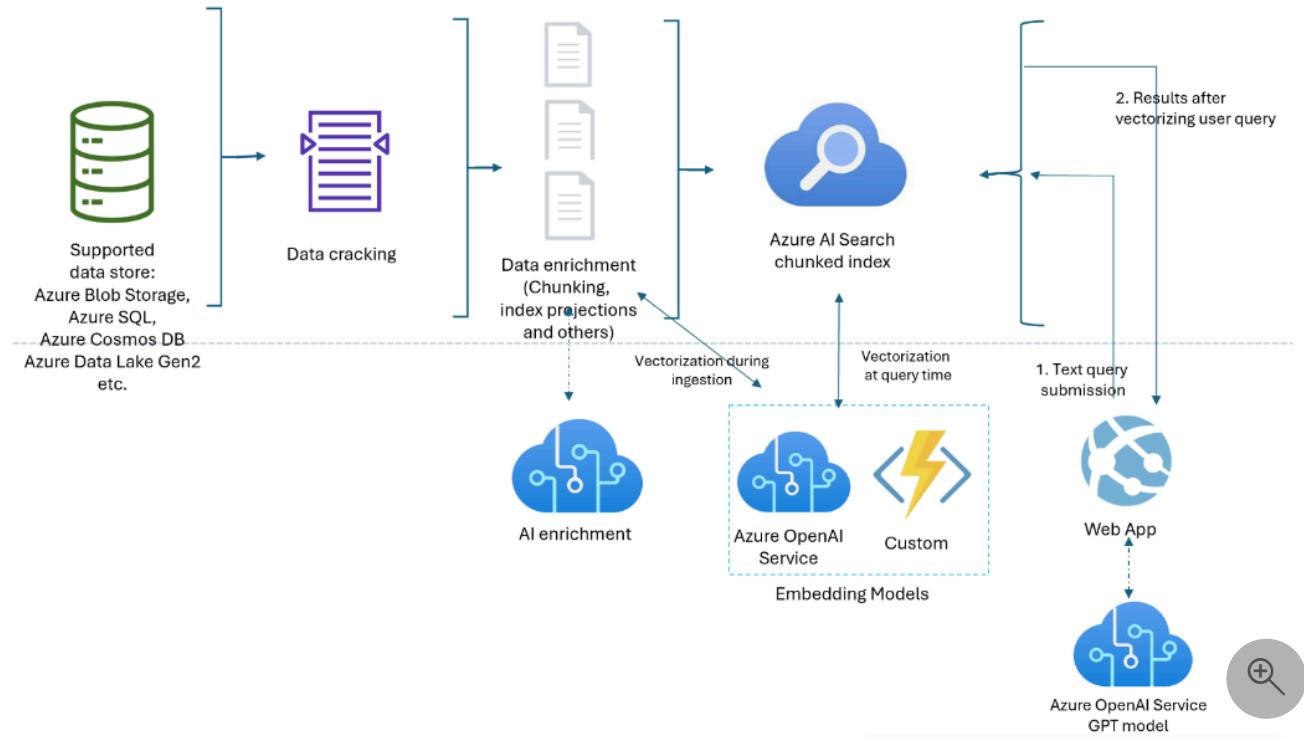
[] [Expand table](#)

Embedding skill	Vectorizer
AzureOpenAIEmbedding skill	Azure OpenAI vectorizer
Custom skill	Custom Web API vectorizer
Azure Vision multimodal embeddings skill (preview)	Azure Vision vectorizer
AML skill pointing to the model catalog in Foundry portal	Microsoft Foundry model catalog vectorizer

Component diagram

The following diagram shows the components of integrated vectorization.

Azure AI Search processing



The workflow is an indexer pipeline. Indexers retrieve data from supported data sources and initiate data enrichment (or applied AI) by calling Azure OpenAI or Foundry Tools or custom code for text-to-vector conversions or other processing.

The diagram focuses on integrated vectorization, but your solution isn't limited to this list. You can add more skills for AI enrichment, create a knowledge store, add semantic ranking, add relevance tuning, and other query features.

Availability and pricing

Integrated vectorization is available in all regions and tiers. However, if you're using skills and vectorizers for AI enrichment, regional requirements might apply. For more information, see [Attach a Foundry resource to a skillset](#).

If you're using a custom skill and an Azure hosting mechanism (such as an Azure function app, Azure Web App, and Azure Kubernetes), check the [Azure product by region page](#) for feature availability.

Data chunking (Text Split skill) is free and available on all Foundry Tools in all regions.

! Note

Some older search services created before January 1, 2019 are deployed on infrastructure that doesn't support vector workloads. If you try to add a vector field to a schema and get

an error, it's a result of outdated services. In this situation, you must create a new search service to try out the vector feature.

What scenarios can integrated vectorization support?

- Subdivide large documents into chunks, useful for vector and nonvector scenarios. For vectors, chunks help you meet the input constraints of embedding models. For nonvector scenarios, you might have a chat-style search app where GPT is assembling responses from indexed chunks. You can use vectorized or nonvectorized chunks for chat-style search.
- Build a vector store where all of the fields are vector fields, and the document ID (required for a search index) is the only string field. Query the vector store to retrieve document IDs, and then send the document's vector fields to another model.
- Combine vector and text fields for hybrid search, with or without semantic ranking. Integrated vectorization simplifies all of the [scenarios supported by vector search](#).

How to use integrated vectorization

For query-only vectorization:

1. [Add a vectorizer](#) to an index. It should be the same embedding model used to generate vectors in the index.
2. [Assign the vectorizer](#) to a vector profile, and then assign a vector profile to the vector field.
3. [Formulate a vector query](#) that specifies the text string to vectorize.

A more common scenario - data chunking and vectorization during indexing:

1. [Create a data source](#) connection to a supported data source for indexer-based indexing.
2. [Create a skillset](#) that calls [Text Split skill](#) for chunking and [Azure OpenAI Embedding](#) or another embedding skill to vectorize the chunks.
3. [Create an index](#) that specifies a [vectorizer](#) for query time, and assign it to vector fields.
4. [Create an indexer](#) to drive everything, from data retrieval, to skillset execution, through indexing. We recommend running the indexer [on a schedule](#) to pick up changed documents or any documents that were missed due to throttling.

Optionally, [create secondary indexes](#) for advanced scenarios where chunked content is in one index, and nonchunked in another index. Chunked indexes (or secondary indexes) are useful

for RAG apps.

💡 Tip

[Try the Import data \(new\) wizard](#) in the Azure portal to explore integrated vectorization before writing any code.

Secure connections to vectorizers and models

If your architecture requires private connections that bypass the internet, you can create a [shared private link connection](#) to the embedding models used by skills during indexing and vectorizers at query time.

Shared private links only work for Azure-to-Azure connections. If you're connecting to OpenAI or another external model, the connection must be over the public internet.

For vectorization scenarios, you would use:

- `openai_account` for embedding models hosted on an Azure OpenAI resource.
- `sites` for embedding models accessed as a [custom skill](#) or [custom vectorizer](#). The `sites` group ID is for App services and Azure functions, which you could use to host an embedding model that isn't one of the Azure OpenAI embedding models.

Benefits

Here are some of the key benefits of the integrated vectorization:

- No separate data chunking and vectorization pipeline. Code is simpler to write and maintain.
- Automate indexing end-to-end. When data changes in the source (such as in Azure Storage, Azure SQL, or Cosmos DB), the indexer can move those updates through the entire pipeline, from retrieval, to document cracking, through optional AI-enrichment, data chunking, vectorization, and indexing.
- Batching and retry logic is built in (non-configurable). Azure AI Search has internal retry policies for throttling errors that surface due to the Azure OpenAI endpoint maxing out on token quotas for the embedding model. We recommend putting the indexer on a schedule (for example, every 5 minutes) so the indexer can process any calls that are throttled by the Azure OpenAI endpoint despite of the retry policies.

- Projecting chunked content to secondary indexes. Secondary indexes are created as you would any search index (a schema with fields and other constructs), but they're populated in tandem with a primary index by an indexer. Content from each source document flows to fields in primary and secondary indexes during the same indexing run.

Secondary indexes are intended for question and answer or chat style apps. The secondary index contains granular information for more specific matches, but the parent index has more information and can often produce a more complete answer. When a match is found in the secondary index, the query returns the parent document from the primary index. For example, assuming a large PDF as a source document, the primary index might have basic information (title, date, author, description), while a secondary index has chunks of searchable content.

Limitations

Make sure you know the [Azure OpenAI quotas and limits for embedding models](#). Azure AI Search has retry policies, but if the quota is exhausted, retries fail.

Azure OpenAI token-per-minute limits are per model, per subscription. Keep this in mind if you're using an embedding model for both query and indexing workloads. If possible, [follow best practices](#). Have an embedding model for each workload, and try to deploy them in different subscriptions.

On Azure AI Search, remember there are [service limits](#) by tier and workloads.

Next steps

- [Set up integrated vectorization](#)
- [Configure a vectorizer in a search index](#)
- [Configure index projections in a skillset](#)

Last updated on 11/18/2025

Agentic retrieval in Azure AI Search

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

What is agentic retrieval? In Azure AI Search, *agentic retrieval* is a new multi-query pipeline designed for complex questions posed by users or agents in chat and copilot apps. It's intended for [Retrieval Augmented Generation \(RAG\)](#) patterns and agent-to-agent workflows.

Here's what it does:

- Uses a large language model (LLM) to break down a complex query into smaller, focused subqueries for better coverage over your indexed content. Subqueries can include chat history for extra context.
- Runs subqueries in parallel. Each subquery is semantically reranked to promote the most relevant matches.
- Combines the best results into a unified response that an LLM can use to generate answers with your proprietary content.
- The response is modular yet comprehensive in how it also includes a query plan and source documents. You can choose to use just the search results as grounding data, or invoke the LLM to formulate an answer.

This high-performance pipeline helps you generate high quality grounding data (or an answer) for your chat application, with the ability to answer complex questions quickly.

Programmatically, agentic retrieval is supported through a new [Knowledge Base object](#) in the 2025-11-01-preview and in Azure SDK preview packages that provide the feature. A knowledge base's retrieval response is designed for downstream consumption by other agents and chat apps.

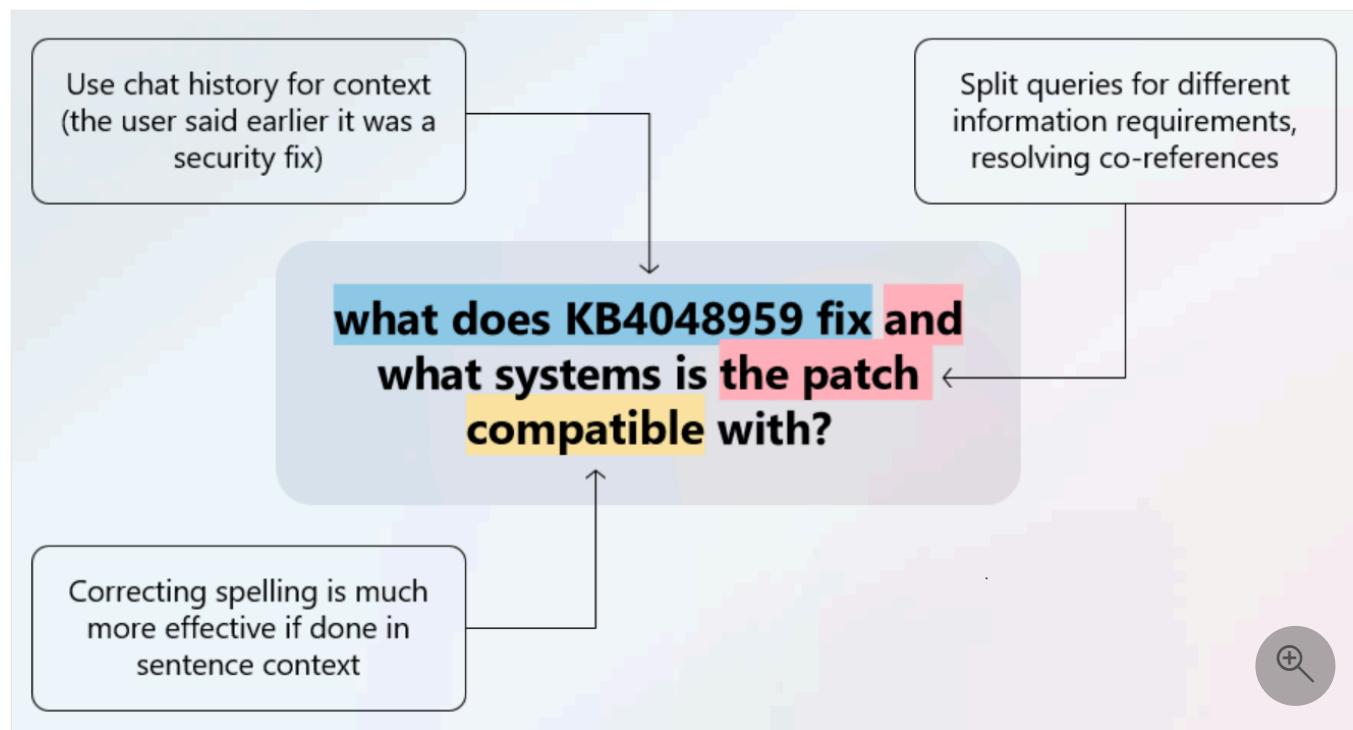
Why use agentic retrieval

You should use agentic retrieval when you want to provide agents and apps with the most relevant content for answering harder questions, leveraging chat context and your proprietary content.

The *agentic* aspect is a reasoning step in query planning processing that's performed by a supported large language model (LLM) that you provide. The LLM analyzes the entire chat thread to identify the underlying information need. Instead of a single, catch-all query, the LLM breaks down compound questions into focused subqueries based on: user questions, chat history, and parameters on the request. The subqueries target your indexed documents (plain text and vectors) in Azure AI Search. This hybrid approach ensures you surface both keyword matches and semantic similarities at once, dramatically improving recall.

The *retrieval* component is the ability to run subqueries simultaneously, merge results, semantically rank results, and return a three-part response that includes grounding data for the next conversation turn, reference data so that you can inspect the source content, and an activity plan that shows query execution steps.

Query expansion and parallel execution, plus the retrieval response, are the key capabilities of agentic retrieval that make it the best choice for generative AI (RAG) applications.



Agentic retrieval adds latency to query processing, but it makes up for it by adding these capabilities:

- Reads in chat history as an input to the retrieval pipeline.
- Deconstructs a complex query that contains multiple "asks" into component parts. For example: "find me a hotel near the beach, with airport transportation, and that's within walking distance of vegetarian restaurants."
- Rewrites an original query into multiple subqueries using synonym maps (optional) and LLM-generated paraphrasing.
- Corrects spelling mistakes.
- Executes all subqueries simultaneously.

- Outputs a unified result as a single string. Alternatively, you can extract parts of the response for your solution. Metadata about query execution and reference data is included in the response.

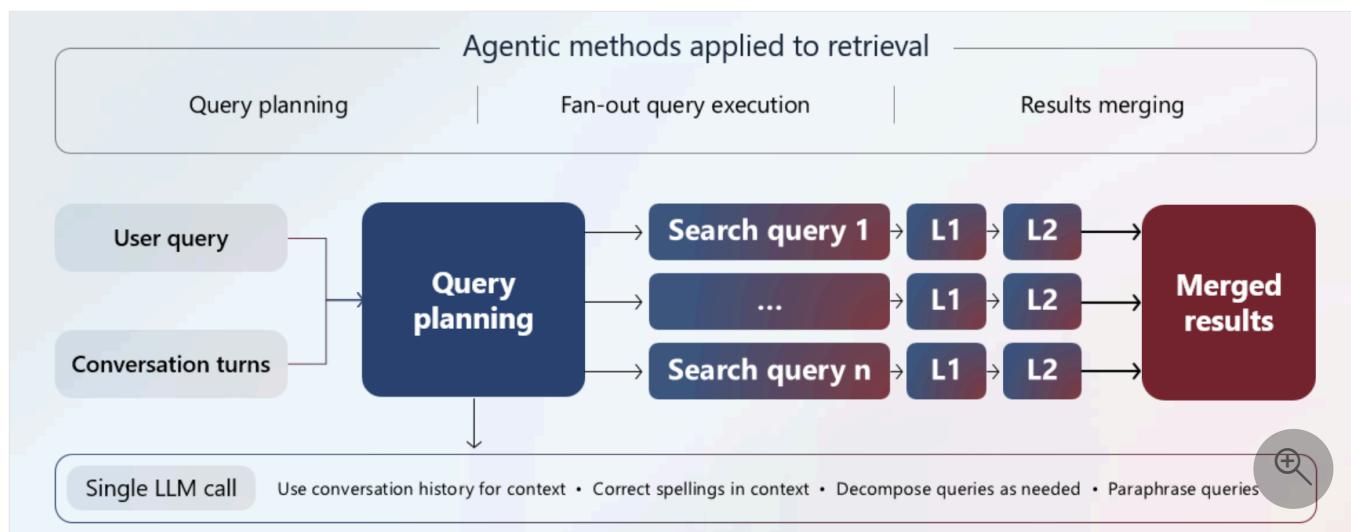
Agentic retrieval invokes the entire query processing pipeline multiple times for each subquery, but it does so in parallel, preserving the efficiency and performance necessary for a reasonable user experience.

(!) Note

Including an LLM in query planning adds latency to a query pipeline. You can mitigate the effects by using faster models, such as gpt-4o-mini, and summarizing the message threads. You can minimize latency and costs by setting properties that limit LLM processing. You can also exclude LLM processing altogether for just text and hybrid search and your own query planning logic.

Architecture and workflow

Agentic retrieval is designed for conversational search experiences that use an LLM to intelligently break down complex queries. The system coordinates multiple Azure services to deliver comprehensive search results.



How it works

The agentic retrieval process works as follows:

1. **Workflow initiation:** Your application calls a knowledge base with retrieve action that provides a query and conversation history.

2. **Query planning:** A knowledge base sends your query and conversation history to an LLM, which analyzes the context and breaks down complex questions into focused subqueries. This step is automated and not customizable.
3. **Query execution:** The knowledge base sends the subqueries to your knowledge sources. All subqueries run simultaneously and can be keyword, vector, and hybrid search. Each subquery undergoes semantic reranking to find the most relevant matches. References are extracted and retained for citation purposes.
4. **Result synthesis:** The system combines all results into a unified response with three parts: merged content, source references, and execution details.

Your search index determines query execution and any optimizations that occur during query execution. Specifically, if your index includes searchable text and vector fields, a hybrid query executes. If the only searchable field is a vector field, then only pure vector search is used. The index semantic configuration, plus optional scoring profiles, synonym maps, analyzers, and normalizers (if you add filters) are all used during query execution. You must have named defaults for a semantic configuration and a scoring profile.

Required components

[] [Expand table](#)

Component	Service	Role
LLM	Azure	Creates subqueries from conversation context and later uses
	OpenAI	grounding data for answer generation
Knowledge base	Azure AI Search	Orchestrates the pipeline, connecting to your LLM and managing query parameters
Knowledge source	Azure AI Search	Wraps the search index with properties pertaining to knowledge base usage
Search index	Azure AI Search	Stores your searchable content (text and vectors) with semantic configuration
Semantic ranker	Azure AI Search	Required component that reranks results for relevance (L2 reranking)

Integration requirements

Your application drives the pipeline by calling the knowledge base and handling the response. The pipeline returns grounding data that you pass to an LLM for answer generation in your

conversation interface. For implementation details, see [Tutorial: Build an end-to-end agentic retrieval solution](#).

! Note

Only gpt-4o, gpt-4.1, and gpt-5 series models are supported for query planning. You can use any model for final answer generation.

How to get started

To create an agentic retrieval solution, you must use a preview REST API version or a prerelease Azure SDK package that provides the functionality.

Currently, Azure portal support for agentic retrieval is limited to the 2025-08-01-preview. We recommend using a programmatic approach to access the latest features.

Quickstarts

- [Quickstart: Use agentic retrieval in Azure AI Search](#)
 - This quickstart supports C#, Java, JavaScript, Python, TypeScript, and REST.
- [Quickstart: Use agentic retrieval in the Azure portal](#)
 - The portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. Breaking changes apply to objects created in this quickstart.

Availability and pricing

Agentic retrieval is available in [selected regions](#). Knowledge sources and knowledge bases also have [maximum limits](#) that vary by service tier.

It has a dependency on premium features. If you disable semantic ranker for your search service, you effectively disable agentic retrieval.

[] [Expand table](#)

Plan	Description
Free	A free tier search service provides 50 million free agentic reasoning tokens per month. On higher tiers, you can choose between the free plan (default) and the standard plan.

Plan	Description
Standard	The standard plan is pay-as-you-go pricing once the monthly free quota is consumed. After the free quota is used up, you are charged an additional fee for each additional one million agentic reasoning tokens. You aren't notified when the transition occurs. For more information about charges by currency, see the Azure AI Search pricing page .

Token-based billing for LLM-based query planning and [answer synthesis](#) (optional) is pay-as-you-go in Azure OpenAI. It's token based for both input and output tokens. The model you assign to the knowledge base is the one [charged for token usage](#). For example, if you use gpt-4o, the token charge appears in the bill for gpt-4o.

Token-based billing for agentic retrieval is the number of tokens returned by each subquery.

 [Expand table](#)

Aspect	Classic single-query pipeline	Agentic retrieval multi-query pipeline
Unit	Query based (1,000 queries) per unit of currency	Token based (1 million tokens per unit of currency)
Cost per unit	Uniform cost per query	Uniform cost per token
Cost estimation	Estimate query count	Estimate token usage
Free tier	1,000 free queries	50 million free tokens

Example: Estimate costs

Agentic retrieval has two billing models: billing from Azure OpenAI (query planning and, if enabled, answer synthesis) and billing from Azure AI Search for agentic retrieval.

This pricing example omits answer synthesis, but helps illustrate the estimation process. Your costs could be lower. For the actual price of transactions, see [Azure OpenAI pricing](#).

Estimated billing costs for query planning

To estimate the query plan costs as pay-as-you-go in Azure OpenAI, let's assume gpt-4o-mini:

- 15 cents for 1 million input tokens.
- 60 cents for 1 million output tokens.
- 2,000 input tokens for average chat conversation size.
- 350 tokens for average output plan size.

Estimated billing costs for query execution

To estimate agentic retrieval token counts, start with an idea of what an average document in your index looks like. For example, you might approximate:

- 10,000 chunks, where each chunk is one to two paragraphs of a PDF.
- 500 tokens per chunk.
- Each subquery reranks up to 50 chunks.
- On average, there are three subqueries per query plan.

Calculating price of execution

1. Assume we make 2,000 agentic retrievals with three subqueries per plan. This gives us about 6,000 total queries.
2. Rerank 50 chunks per subquery, which is 300,000 total chunks.
3. Average chunk is 500 tokens, so the total tokens for reranking is 150 million.
4. Given a hypothetical price of 0.022 per token, \$3.30 is the total cost for reranking in US dollars.
5. Moving on to query plan costs: 2,000 input tokens multiplied by 2,000 agentic retrievals equal 4 million input tokens for a total of 60 cents.
6. Estimate the output costs based on an average of 350 tokens. If we multiply 350 by 2,000 agentic retrievals, we get 700,000 output tokens total for a total of 42 cents.

Putting it all together, you'd pay about \$3.30 for agentic retrieval in Azure AI Search, 60 cents for input tokens in Azure OpenAI, and 42 cents for output tokens in Azure OpenAI, for \$1.02 for query planning total. The combined cost for the full execution is \$4.32.

Last updated on 11/18/2025

Full text search in Azure AI Search

Article • 03/07/2025

Full text search is an approach in information retrieval that matches on plain text stored in an index. For example, given a query string "hotels in San Diego on the beach", the search engine looks for tokenized strings based on those terms. To make scans more efficient, query strings undergo lexical analysis: lower-casing all terms, removing stop words like "the", and reducing terms to primitive root forms. When matching terms are found, the search engine retrieves documents, ranks them in order of relevance, and returns the top results.

Query execution can be complex. This article is for developers who need a deeper understanding of how full text search works in Azure AI Search. For text queries, Azure AI Search seamlessly delivers expected results in most scenarios, but occasionally, you might get a result that seems "off" somehow. In these situations, having a background in the four stages of Lucene query execution (query parsing, lexical analysis, document matching, and scoring) can help you identify specific changes to query parameters or index configuration that produce the desired outcome.

ⓘ Note

Azure AI Search uses [Apache Lucene](#) for full text search, but Lucene integration isn't exhaustive. We selectively expose and extend Lucene functionality to enable the scenarios important to Azure AI Search.

Architecture overview and diagram

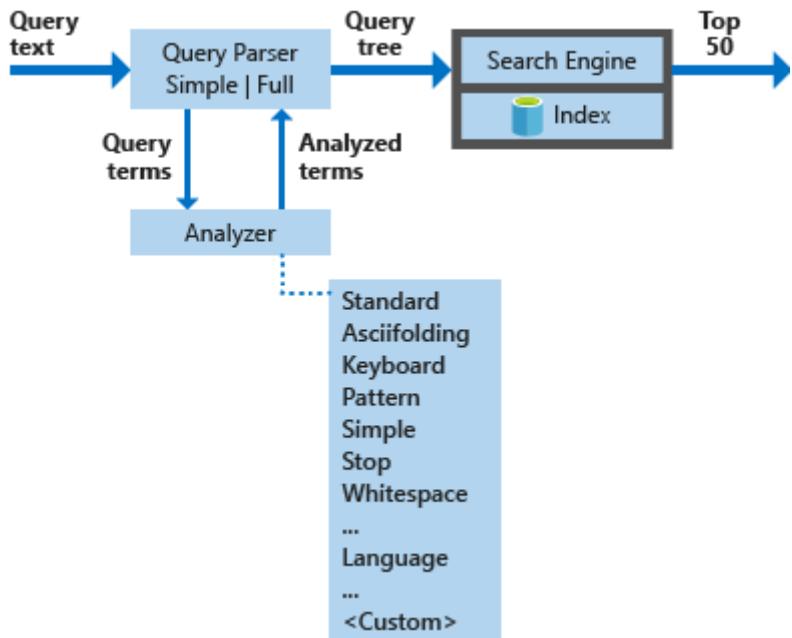
Query execution has four stages:

1. Query parsing
2. Lexical analysis
3. Document retrieval
4. Scoring

A full text search query starts with parsing the query text to extract search terms and operators. There are two parsers so that you can choose between speed and complexity. An analysis phase is next, where individual query terms are sometimes broken down and reconstituted into new forms. This step helps to cast a broader net over what could be considered as a potential match. The search engine then scans the index to find documents with matching terms and scores each match. A result set is then sorted by a

relevance score assigned to each individual matching document. Those at the top of the ranked list are returned to the calling application.

The following diagram illustrates the components used to process a search request:



[] Expand table

Key components	Functional description
Query parsers	Separate query terms from query operators and create the query structure (a query tree) to be sent to the search engine.
Analyzers	Perform lexical analysis on query terms. This process can involve transforming, removing, or expanding query terms.
Index	An efficient data structure used to store and organize searchable terms extracted from indexed documents.
Search engine	Retrieves and scores matching documents based on the contents of the inverted index.

Anatomy of a search request

A search request is a complete specification of what should be returned in a result set. In its simplest form, it's an empty query with no criteria of any kind. A more realistic example includes parameters, several query terms, perhaps scoped to certain fields, with possibly a filter expression and ordering rules.

The following example is a search request you might send to Azure AI Search using the [REST API](#):

```
POST /indexes/hotels/docs/search?api-version=2024-07-01
{
  "search": "Spacious, air-condition* +\"Ocean view\"",
  "searchFields": "description, title",
  "searchMode": "any",
  "filter": "price ge 60 and price lt 300",
  "orderby": "geo.distance(location, geography'POINT(-159.476235
22.227659)'),",
  "queryType": "full"
}
```

For this request, the search engine does the following operations:

1. Finds documents where the price is at least \$60 and less than \$300.
2. Executes the query. In this example, the search query consists of phrases and terms: "Spacious, air-condition* +\"Ocean view\"" (Users typically don't enter punctuation, but by including it in the example, we can explain how analyzers handle it.)

For this query, the search engine scans the description and title fields specified in "searchFields" for documents that contain "Ocean view", and additionally on the term "spacious", or on terms that start with the prefix "air-condition". The "searchMode" parameter is used to match on any term (default) or all of them, for cases where a term isn't explicitly required (+).

3. Orders the resulting set of hotels by proximity to a given geography location, and then returns the results to the calling application.

Most of this article is about processing of the *search query*: "Spacious, air-condition* +\"Ocean view\"". Filtering and ordering are out of scope. For more information, see the [Search API reference documentation](#).

Stage 1: Query parsing

As noted, the query string is the first line of the request:

```
"search": "Spacious, air-condition* +\"Ocean view\"",
```

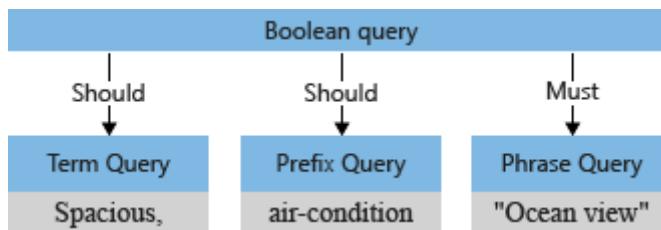
The query parser separates operators (such as `*` and `+` in the example) from search terms and deconstructs the search query into *subqueries* of a supported type:

- *term query* for standalone terms (like `spacious`)
- *phrase query* for quoted terms (like `ocean view`)
- *prefix query* for terms followed by a prefix operator `*` (like `air-condition`)

For a full list of supported query types, see [Lucene query syntax](#).

Operators associated with a subquery determine whether the query "must be" or "should be" satisfied in order for a document to be considered a match. For example, `+"Ocean view"` is "must" due to the `+` operator.

The query parser restructures the subqueries into a *query tree* (an internal structure representing the query), which it passes to the search engine. In the first stage of query parsing, the query tree looks like this:



Supported parsers: Simple and Full Lucene

Azure AI Search exposes two different query languages: `simple` (default) and `full`. By setting the `queryType` parameter with your search request, you tell the query parser which query language you choose so that it knows how to interpret the operators and syntax.

- The [Simple query language](#) is intuitive and robust, often suitable for interpreting user input as-is without client-side processing. It supports query operators familiar from web search engines.
- The [Full Lucene query language](#), which you get by setting `queryType=full`, extends the default Simple query language by adding support for more operators and query types like wildcard, fuzzy, regex, and field-scoped queries. For example, a regular expression sent in Simple query syntax would be interpreted as a query string and not an expression. The example request in this article uses the Full Lucene query language.

Impact of searchMode on the parser

Another search request parameter that affects parsing is the "searchMode" parameter. It controls the default operator for Boolean queries: any (default) or all.

When "searchMode=any", which is the default, the space delimiter between spacious and air-condition is OR (||), making the sample query text equivalent to:

```
Spacious, ||air-condition*+"Ocean view"
```

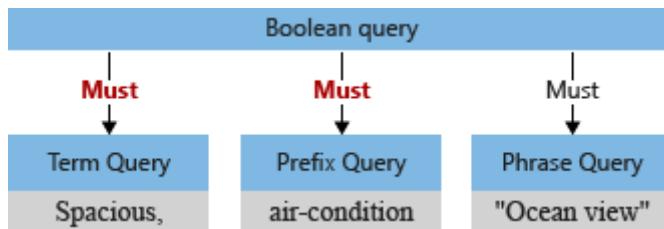
Explicit operators, such as + in +"Ocean view", are unambiguous in boolean query construction (the term *must* match). Less obvious is how to interpret the remaining terms: spacious and air-condition. Should the search engine find matches on ocean view *and* spacious *and* air-condition? Or should it find ocean view plus *either one* of the remaining terms?

By default ("searchMode=any"), the search engine assumes the broader interpretation. Either field *should* be matched, reflecting "or" semantics. The initial query tree illustrated previously, with the two "should" operations, shows the default.

Suppose that we now set "searchMode=all". In this case, the space is interpreted as an "and" operation. Both of the remaining terms must be present in the document to qualify as a match. The resulting sample query would be interpreted like this:

```
+Spacious,+air-condition*+"Ocean view"
```

A modified query tree for this query, where a matching document is the intersection of all three subqueries, would look like this:



① Note

Choosing "searchMode=any" over "searchMode=all" is a decision best made by running representative queries. Users who are likely to include operators (common when searching document stores) might find results more intuitive if

"searchMode=all" informs boolean query constructs. For more information about the interplay between "searchMode" and operators, see [Simple query syntax](#).

Stage 2: Lexical analysis

Lexical analyzers process *term queries* and *phrase queries* after the query tree is structured. An analyzer accepts the text inputs given to it by the parser, processes the text, and then sends back tokenized terms to be incorporated into the query tree.

The most common form of lexical analysis is *linguistic analysis*, which transforms query terms based on rules specific to a given language. This involves:

- Reducing a query term to the root form of a word.
- Removing non-essential words ([stopwords](#), such as "the" or "and" in English).
- Breaking a composite word into component parts.
- Lowercasing an uppercase word.

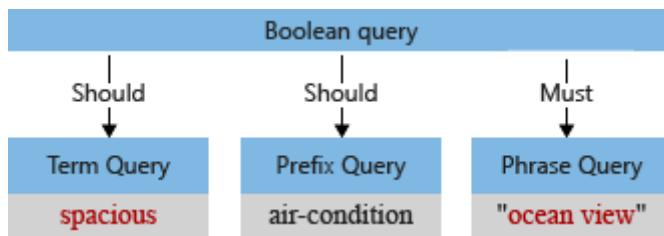
All of these operations tend to erase differences between the text input provided by the user and the terms stored in the index. Such operations go beyond text processing and require in-depth knowledge of the language itself. To add this layer of linguistic awareness, Azure AI Search supports a long list of [language analyzers](#) from both Lucene and Microsoft.

Note

Depending on your scenario, analysis requirements can range from minimal to elaborate. You can control the complexity of lexical analysis by selecting one of the predefined analyzers or by creating your own [custom analyzer](#). Analyzers are scoped to searchable fields and are specified as part of a field definition. This allows you to vary lexical analysis on a per-field basis. If unspecified, the *standard* Lucene analyzer is used.

In our example, prior to analysis, the initial query tree has the term "Spacious," with an uppercase "S" and a comma that the query parser interprets as a part of the query term (a comma isn't considered a query language operator).

When the default analyzer processes the term, it will lowercase "ocean view" and "spacious" and remove the comma character. The modified query tree looks like this:



Testing analyzer behaviors

The behavior of an analyzer can be tested using the [Analyze API](#). Provide the text you want to analyze to see what terms the given analyzer generates. For example, to see how the standard analyzer would process the text "air-condition", you can issue the following request:

JSON

```
{
  "text": "air-condition",
  "analyzer": "standard"
}
```

The standard analyzer breaks the input text into the following two tokens, annotating them with attributes like start and end offsets (used for hit highlighting) as well as their position (used for phrase matching):

JSON

```
{
  "tokens": [
    {
      "token": "air",
      "startOffset": 0,
      "endOffset": 3,
      "position": 0
    },
    {
      "token": "condition",
      "startOffset": 4,
      "endOffset": 13,
      "position": 1
    }
  ]
}
```

Exceptions to lexical analysis

Lexical analysis applies only to query types that require complete terms, either a term query or a phrase query. It doesn't apply to query types with incomplete terms—prefix query, wildcard query, and regex query—or to a fuzzy query. Those query types, including the prefix query with the term `air-condition*` in our example, are added directly to the query tree, bypassing the analysis stage. The only transformation performed on query terms of those types is lowercasing.

Stage 3: Document retrieval

Document retrieval refers to finding documents with matching terms in the index. This stage is best understood through an example. Let's start with a hotels index that has the following simple schema:

```
JSON

{
    "name": "hotels",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "title", "type": "Edm.String", "searchable": true },
        { "name": "description", "type": "Edm.String", "searchable": true }
    ]
}
```

Further assume that this index contains the following four documents:

```
JSON

{
    "value": [
        {
            "id": "1",
            "title": "Hotel Atman",
            "description": "Spacious rooms, ocean view, walking distance to
the beach."
        },
        {
            "id": "2",
            "title": "Beach Resort",
            "description": "Located on the north shore of the island of
Kaua'i. Ocean view."
        },
        {
            "id": "3",
            "title": "Playa Hotel",
            "description": "Comfortable, air-conditioned rooms with ocean
view."
        }
    ]
}
```

```
    },
    {
        "id": "4",
        "title": "Ocean Retreat",
        "description": "Quiet and secluded"
    }
]
```

How terms are indexed

To understand retrieval, it helps to know a few basics about indexing. The unit of storage is an inverted index, one for each searchable field. Within an inverted index is a sorted list of all terms from all documents. Each term maps to the list of documents in which it occurs, as evident in the example below.

To produce the terms in an inverted index, the search engine performs lexical analysis over the content of documents, similar to what happens during query processing:

1. *Text inputs* are passed to an analyzer, lowercased, stripped of punctuation, and so forth, depending on the analyzer configuration.
2. *Tokens* are the output of lexical analysis.
3. *Terms* are added to the index.

It's common, but not required, to use the same analyzers for search and indexing operations so that query terms look more like terms inside the index.

ⓘ Note

Azure AI Search lets you specify different analyzers for indexing and search via additional `indexAnalyzer` and `searchAnalyzer` field parameters. If unspecified, the analyzer set with the `analyzer` property is used for both indexing and searching.

Inverted index for example documents

Returning to our example, for the `title` field, the inverted index looks like this:

[+] [Expand table](#)

Term	Document list
atman	1

Term	Document list
beach	2
hotel	1, 3
ocean	4
playa	3
resort	3
retreat	4

In the title field, only *hotel* shows up in two documents: 1 and 3.

For the **description** field, the index looks like this:

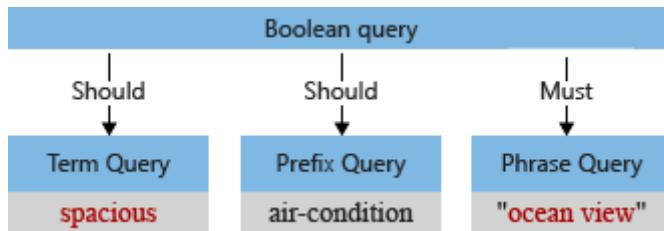
[\[\] Expand table](#)

Term	Document list
air	3
and	4
beach	1
conditioned	3
comfortable	3
distance	1
island	2
kaua'i	2
located	2
north	2
ocean	1, 2, 3
of	2
on	2
quiet	4
rooms	1, 3

Term	Document list
secluded	4
shore	2
spacious	1
the	1, 2
to	1
view	1, 2, 3
walking	1
with	3

Matching query terms against indexed terms

Given the inverted indexes above, let's return to the sample query and see how matching documents are found for our example query. Recall that the final query tree looks like this:



During query execution, individual queries are executed against the searchable fields independently.

- The TermQuery, "spacious", matches document 1 (Hotel Atman).
- The PrefixQuery, "air-condition*", doesn't match any documents.

This behavior sometimes confuses developers. Although the term air-conditioned exists in the document, it's split into two terms by the default analyzer. Recall that prefix queries, which contain partial terms, aren't analyzed. Therefore, terms with the prefix "air-condition" are looked up in the inverted index and not found.

- The PhraseQuery, "ocean view", looks up the terms "ocean" and "view" and checks the proximity of terms in the original document. Documents 1, 2, and 3 match this query in the description field. Notice document 4 has the term "ocean" in the title but isn't considered a match, as we're looking for the "ocean view" phrase rather than individual words.

Note

A search query is executed independently against all searchable fields in the Azure AI Search index unless you limit the fields set with the `searchFields` parameter, as illustrated in the example search request. Documents that match in any of the selected fields are returned.

On the whole, for the query in question, the documents that match are 1, 2, and 3.

Stage 4: Scoring

Every document in a search result set is assigned a relevance score. The function of the relevance score is to rank higher those documents that best answer a user question as expressed by the search query. The score is computed based on statistical properties of terms that matched. At the core of the scoring formula is [term frequency-inverse document frequency](#) (TF/IDF). In queries containing rare and common terms, TF/IDF promotes results containing the rare term. For example, in a hypothetical index with all Wikipedia articles, from documents that matched the query *the president*, documents matching on *president* are considered more relevant than documents matching on *the*.

Scoring example

Recall the three documents that matched our example query:

```
search=Spacious, air-condition* +"Ocean view"
```

JSON

```
{
  "value": [
    {
      "@search.score": 0.25610128,
      "id": "1",
      "title": "Hotel Atman",
      "description": "Spacious rooms, ocean view, walking distance to the beach."
    },
    {
      "@search.score": 0.08951007,
      "id": "3",
      "title": "Playa Hotel",
      "description": "Comfortable, air-conditioned rooms with ocean view."
```

```
        },
        {
            "@search.score": 0.05967338,
            "id": "2",
            "title": "Ocean Resort",
            "description": "Located on a cliff on the north shore of the island of Kauai. Ocean view."
        }
    ]
}
```

Document 1 matched the query best because both the term *spacious* and the required phrase *ocean view* occur in the description field. The next two documents match only the phrase *ocean view*. You might be surprised that the relevance scores for documents 2 and 3 are different, even though they matched the query in the same way. That's because the scoring formula has more components than just TF/IDF. In this case, document 3 was assigned a slightly higher score because its description is shorter. Learn about [Lucene's Practical Scoring Formula](#) ↗ to understand how field length and other factors can influence the relevance score.

Some query types (wildcard, prefix, and regex) always contribute a constant score to the overall document score. This allows matches found through query expansion to be included in the results without affecting the ranking.

An example illustrates why this matters. Wildcard searches, including prefix searches, are ambiguous by definition because the input is a partial string with potential matches on a very large number of disparate terms. Consider an input of "tour*", with matches found on "tours", "tourettes", and "tourmaline". Given the nature of these results, there's no way to reasonably infer which terms are more valuable than others. For this reason, we ignore term frequencies when scoring results in queries of types wildcard, prefix, and regex. In a multi-part search request that includes partial and complete terms, results from the partial input are incorporated with a constant score to avoid bias towards potentially unexpected matches.

Relevance tuning

There are two ways to tune relevance scores in Azure AI Search:

- **Scoring profiles** promote documents in the ranked list of results based on a set of rules. In our example, we could consider documents that matched in the title field more relevant than documents that matched in the description field. Additionally, if our index had a price field for each hotel, we could promote documents with lower prices. Learn more about [adding scoring profiles to a search index](#).

- **Term boosting** (available only in the Full Lucene query syntax) provides a boosting operator `^` that can be applied to any part of the query tree. In our example, instead of searching on the prefix `air-condition*`, one could search for either the exact term `air-condition` or the prefix, but documents that match on the exact term are ranked higher by applying boost to the term query: `air-condition^2||air-condition*`. Learn more about [term boosting in a query](#).

Scoring in a distributed index

All indexes in Azure AI Search are automatically split into multiple shards, allowing us to quickly distribute the index among multiple nodes during service scale up or scale down. When a search request is issued, it's issued against each shard independently. The results from each shard are then merged and ordered by score (if no other ordering is defined). It's important to know that the scoring function weights query term frequency against its inverse document frequency in all documents within the shard, not across all shards!

This means a relevance score *could* be different for identical documents if they reside on different shards. Fortunately, such differences tend to disappear as the number of documents in the index grows due to more even term distribution. It's not possible to assume on which shard any given document will be placed. However, assuming a document key doesn't change, it will always be assigned to the same shard.

In general, document score isn't the best attribute for ordering documents if order stability is important. For example, given two documents with an identical score, there's no guarantee that one appears first in subsequent runs of the same query. Document score should only give a general sense of document relevance relative to other documents in the results set.

Conclusion

The success of commercial search engines has raised expectations for full text search over private data. For almost any kind of search experience, we now expect the engine to understand our intent, even when terms are misspelled or incomplete. We might even expect matches based on near equivalent terms or synonyms that we never specified.

From a technical standpoint, full text search is highly complex, requiring sophisticated linguistic analysis and a systematic approach to processing in ways that distill, expand, and transform query terms to deliver a relevant result. Given the inherent complexities, there are many factors that can affect the outcome of a query. For this reason, investing

the time to understand the mechanics of full text search offers tangible benefits when trying to work through unexpected results.

This article explored full text search in the context of Azure AI Search. We hope it gives you sufficient background to recognize potential causes and resolutions for addressing common query problems.

Next steps

- Build the sample index, try out different queries, and review results. For instructions, see [Build and query an index in the Azure portal](#).
- Try other query syntax from the [Search Documents](#) example section or from [Simple query syntax](#) in Search explorer in the Azure portal.
- Review [scoring profiles](#) if you want to tune ranking in your search application.
- Apply [language-specific lexical analyzers](#).
- [Configure custom analyzers](#) for either minimal processing or specialized processing on specific fields.

Related content

- [Search Documents REST API](#)
- [Simple query syntax](#)
- [Full Lucene query syntax](#)
- [Handle search results](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Vector search in Azure AI Search

Vector search is an information retrieval approach that supports indexing and querying over numeric representations of content. Because the content is numeric rather than plain text, matching is based on vectors that are most similar to the query vector, which enables matching across:

- Semantic or conceptual likeness. For example, "dog" and "canine" are conceptually similar but linguistically distinct.
- Multilingual content, such as "dog" in English and "hund" in German.
- Multiple content types, such as "dog" in plain text and an image of a dog.

This article provides an overview of vector search in Azure AI Search, including supported scenarios, availability, and integration with other Azure services.

Tip

Want to get started right away? Follow these steps:

1. [Provide embeddings](#) for your index or [generate embeddings](#) in an indexer pipeline.
2. [Create a vector index](#).
3. [Run vector queries](#).

What scenarios can vector search support?

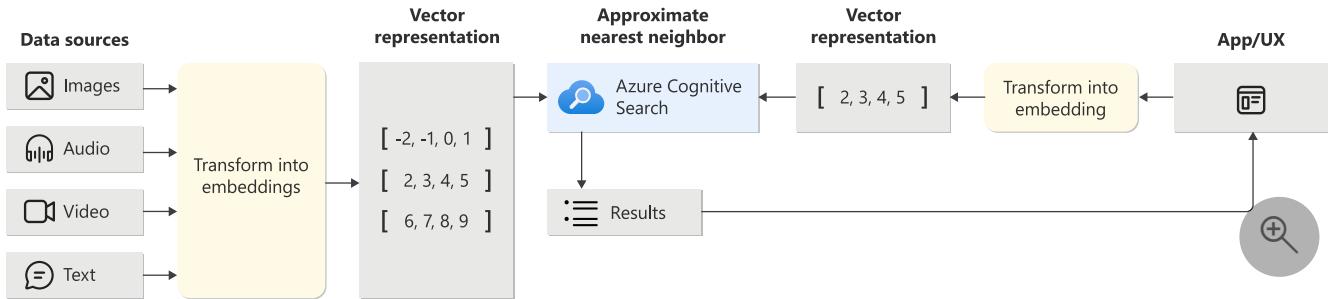
Vector search supports the following scenarios:

- **Similarity search.** Encode text using embedding models or open-source models, such as OpenAI embeddings or SBERT, respectively. You then retrieve documents using queries that are also encoded as vectors.
- **Hybrid search.** Azure AI Search defines hybrid search as the execution of vector search and [keyword search](#) in the same request. Vector support is implemented at the field level. If an index contains vector and nonvector fields, you can write a query that targets both. The queries execute in parallel, and the results are merged into a single response and ranked accordingly.
- **Multimodal search.** Encode text and images using multimodal embeddings, such as [OpenAI CLIP](#) or [GPT-4 Turbo with Vision](#) in Azure OpenAI, and then query an embedding space composed of vectors from both content types.

- **Multilingual search.** Azure AI Search is designed for extensibility. If you have embedding models and chat models trained in multiple languages, you can call them through custom or built-in skills on the indexing side or vectorizers on the query side. For more control over text translation, use the [multi-language capabilities](#) supported by Azure AI Search for nonvector content in hybrid search scenarios.
- **Filtered vector search.** A query request can include a vector query and a [filter expression](#). Filters apply to text and numeric fields. They're useful for metadata filters and for including or excluding search results based on filter criteria. Although a vector field isn't filterable, you can set up a filterable text or numeric field. The search engine can process the filter before or after executing the vector query.
- **Vector database.** Azure AI Search stores the data that you query over. Use it as a [pure vector index](#) when you need long-term memory or a knowledge base, grounding data for the [retrieval-augmented generation \(RAG\)](#) architecture, or an app that uses vectors.

How does vector search work?

Azure AI Search supports indexing, storing, and querying vector embeddings from a search index. The following diagram shows the indexing and query workflows for vector search.



On the indexing side, Azure AI Search uses a [nearest neighbors algorithm](#) to place similar vectors close together in an index. Internally, it creates [vector indexes](#) for each vector field.

How you get embeddings from your source content into Azure AI Search depends on your processing approach:

- For internal processing, Azure AI Search offers [integrated data chunking and vectorization](#) in an indexer pipeline. You provide the necessary resources, such as endpoints and connection information for Azure OpenAI. Azure AI Search then makes the calls and handles the transitions. This approach requires an indexer, a supported data source, and a skillset that drives chunking and embedding.
- For external processing, you can [generate embeddings](#) outside of Azure AI Search and push the prevectorized content directly into [vector fields](#) in your search index.

On the query side, your client app collects user input, typically through a prompt. You can add an encoding step to vectorize the input and then send the vector query to your Azure AI Search index for similarity search. As with indexing, you can use [integrated vectorization](#) to encode the query. For either approach, Azure AI Search returns documents with the requested k nearest neighbors (kNN) in the results.

Azure AI Search supports [hybrid scenarios](#) that run vector and keyword search in parallel and return a unified result set, which often provides better results than vector or keyword search alone. For hybrid search, both vector and nonvector content are ingested into the same index for queries that run simultaneously.

Availability and pricing

Vector search is available in [all regions](#) and on [all tiers](#) at no extra charge. However, generating embeddings or using AI enrichment for vectorization might incur charges from the model provider.

For portal and programmatic access to vector search, you can use:

- The [Import data \(new\) wizard](#) in the Azure portal.
- The [Search Service REST APIs](#).
- The Azure SDKs for [.NET](#), [Python](#), and [JavaScript](#).
- [Other Azure offerings](#), such as Microsoft Foundry.

! Note

- Some search services created before January 1, 2019 don't support vector workloads. If you try to add a vector field to a schema and get an error, it's a result of outdated services. In this situation, you must create a new search service to try out the vector feature.
- Search services created after April 3, 2024 offer [higher quotas for vector indexes](#). If you have an older service, you might be able to [upgrade your service](#) for higher vector quotas.

Azure integration and related services

Azure AI Search is deeply integrated across the Azure AI platform. The following table lists products that are useful in vector workloads.

Product	Integration
Azure OpenAI	Azure OpenAI provides embedding models and chat models. Demos and samples target the text-embedding-ada-002 model. We recommend Azure OpenAI for generating embeddings for text.
Foundry Tools	Image Retrieval Vectorize Image API supports vectorization of image content. We recommend this API for generating embeddings for images.
Foundry Agent Service	In Azure AI Search, you can create an <i>indexed knowledge source</i> that points to a search index containing vector fields and a vectorizer. You can then parent the knowledge source to a <i>knowledge base</i> and connect the knowledge base to Foundry Agent Service , providing your agents with vector search results for enhanced knowledge retrieval.
Azure data platforms: Azure Blob Storage, Azure Cosmos DB, Azure SQL, Microsoft OneLake	You can use indexers to automate data ingestion, and then use integrated vectorization to generate embeddings. Azure AI Search can automatically index vector data from Azure blob indexers , Azure Cosmos DB for NoSQL indexers , Azure Data Lake Storage Gen2 , Azure Table Storage , and Microsoft OneLake . For more information, see Add vector fields to a search index..

It's also commonly used in open-source frameworks like [LangChain](#).

Related content

- [Quickstart: Vector search using REST](#)
- [Create a vector index](#)
- [Create a vector query](#)
- [azure-vector-search-samples](#)
- [Azure Cognitive Search and LangChain: A Seamless Integration for Enhanced Vector Search Capabilities](#)

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

Hybrid search using vectors and full text in Azure AI Search

A hybrid search is a single query request configured for both full-text and vector queries. It executes against a search index that contains searchable, plain-text content and generated embeddings. For query purposes, hybrid search:

- Is a single query request that includes both `search` and `vectors` query parameters.
- Executes full-text search and vector search in parallel.
- Merges results from each query using [Reciprocal Rank Fusion \(RRF\)](#).

This article explains the concepts, benefits, and limitations of hybrid search. Links at the end provide usage instructions and next steps. You can also watch the [embedded video](#) for an explanation of how hybrid retrieval contributes to high-quality generative search applications.

Why use hybrid search?

Hybrid search combines the strengths of vector search and keyword search. The advantage of vector search is finding information that's conceptually similar to your search query, even if there are no keyword matches in the inverted index. The advantage of keyword or full-text search is precision, with the ability to apply optional semantic ranking that improves the quality of the initial results. Some scenarios, such as querying over product codes, highly specialized jargon, dates, and people's names, can perform better with keyword search because it can identify exact matches.

Benchmark testing on real-world and benchmark datasets indicates that hybrid retrieval with semantic ranker offers significant benefits in search relevance.

The following video explains how hybrid retrieval gives you optimal grounding data for generating useful AI responses.

<https://www.youtube-nocookie.com/embed/Xwx1DJ0qCk> ↗

How does hybrid search work?

In a search index, vector fields containing embeddings coexist with textual and numerical fields, allowing you to formulate hybrid queries that execute simultaneously. Hybrid queries take advantage of existing text-based functionality like filtering, facetting, sorting, scoring profiles, and [semantic ranking](#) on your text fields, while executing a similarity search against vectors in a single search request.

Hybrid search combines results from both full-text and vector queries, which use different ranking functions such as BM25 for text, and Hierarchical Navigable Small World (HNSW) and exhaustive K Nearest Neighbors (eKNN) for vectors. An [RRF](#) algorithm merges the results. The query response provides just one result set, using RRF to rank the unified results.

Structure of a hybrid query

Hybrid search relies on a search index that contains fields of various [data types](#), including plain text and numbers, geo coordinates if you want geospatial search, and vectors to mathematically represent a chunk of text. You can use almost all query capabilities in Azure AI Search with a vector query, except for pure text client-side interactions, such as autocomplete and suggestions.

A representative hybrid query might look like the following. For brevity, the vector queries have placeholder values.

HTTP

```
POST https://{{searchServiceName}}.search.windows.net/indexes/hotels-vector-quickstart/docs/search?api-version=2025-09-01
content-type: application/JSON

{
    "count": true,
    "search": "historic hotel walk to restaurants and shopping",
    "select": "HotelId, HotelName, Category, Description, Address/City, Address/StateProvince",
    "filter": "geo.distance(Location, geography'POINT(-77.03241 38.90166)') le 300",
    "vectorFilterMode": "postFilter",
    "facets": [ "Address/StateProvince"],
    "vectorQueries": [
        {
            "kind": "vector",
            "vector": [ <array of embeddings> ]
            "k": 50,
            "fields": "DescriptionVector",
            "exhaustive": true,
            "oversampling": 20
        },
        {
            "kind": "vector",
            "vector": [ <array of embeddings> ]
            "k": 50,
            "fields": "Description_frVector",
            "exhaustive": false,
            "oversampling": 10
        }
    ],
    "skip": 0,
```

```
"top": 10,  
"queryType": "semantic",  
"queryLanguage": "en-us",  
"semanticConfiguration": "my-semantic-config"  
}
```

Key points:

- `search` specifies a single full-text search query.
- `vectorQueries` specifies vector queries, which can be multiple, targeting multiple vector fields. If the embedding space includes multi-lingual content, vector queries can find the match with no language analyzers or translation required. If you're using semantic ranker, set `k` to 50 to maximize its inputs.
- `select` specifies which fields to return in results, which should be human-readable text fields if you're showing them to users or sending them to a large language model (LLM).
- `filters` can specify geospatial search or other inclusion and exclusion criteria, such as whether parking is included. The geospatial query in this example finds hotels within a 300-kilometer radius of Washington D.C. You can apply the filter at the beginning or end of query processing. If you're using semantic ranker, you probably want post-filtering as the last step, but you should test to confirm which behavior is best for your queries.
- `facets` can be used to compute facet buckets over results that are returned from hybrid queries.
- `queryType=semantic` invokes [semantic ranker](#), applying machine reading comprehension to surface more relevant search results. Semantic ranking is optional. If you aren't using this feature, remove the last three lines of the hybrid query.

Filters and facets target data structures within the index that are distinct from the inverted indexes used for full-text search and the vector indexes used for vector search. As such, when filters and faceted operations execute, the search engine can apply the operational result to the hybrid search results in the response.

Notice how there's no `orderby` in the query. Explicit sort orders override relevanced-ranked results, so if you want similarity and BM25 relevance, omit sorting in your query.

A response from the query might look like the following JSON.

JSON

```
{  
  "@odata.count": 3,  
  "@search.facets": {  
    "Address/StateProvince": [  
      {  
        "count": 1,  
        "value": "NY"
```

```

        },
        {
            "count": 1,
            "value": "VA"
        }
    ]
},
"value": [
{
    "@search.score": 0.03333333507180214,
    "@search.rerankerScore": 2.5229012966156006,
    "HotelId": "49",
    "HotelName": "Swirling Currents Hotel",
    "Description": "Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center.",
    "Category": "Luxury",
    "Address": {
        "City": "Arlington",
        "StateProvince": "VA"
    }
},
{
    "@search.score": 0.032522473484277725,
    "@search.rerankerScore": 2.111117362976074,
    "HotelId": "48",
    "HotelName": "Nordick's Valley Motel",
    "Description": "Only 90 miles (about 2 hours) from the nation's capital and nearby most everything the historic valley has to offer. Hiking? Wine Tasting? Exploring the caverns? It's all nearby and we have specially priced packages to help make our B&B your home base for fun while visiting the valley.",
    "Category": "Boutique",
    "Address": {
        "City": "Washington D.C.",
        "StateProvince": null
    }
}
]
}

```

Related content

- [Create a hybrid query](#)
- [Relevance scoring in hybrid search](#)
- [Outperform vector search with hybrid retrieval and ranking \(Tech blog\)](#) ↗

Multimodal search in Azure AI Search

Multimodal search refers to the ability to ingest, understand, and retrieve information across multiple content types, including text, images, video, and audio. In Azure AI Search, multimodal search natively supports the ingestion of documents containing text and images and the retrieval of their content, enabling you to perform searches that combine both modalities.

Building a robust multimodal pipeline typically involves:

1. Extracting inline images and page text from documents.
2. Describing images in natural language.
3. Embedding both text and images into a shared vector space.
4. Storing the images for later use as annotations.

Multimodal search also requires preserving the order of information as it appears in the documents and executing [hybrid queries](#) that combine [full-text search](#) with [vector search](#) and [semantic ranking](#).

In practice, an application that uses multimodal search can answer questions like "What is the process to have an HR form approved?" even when the only authoritative description of the process lives inside an embedded diagram in a PDF file.

Why use multimodal search?

Traditionally, multimodal search requires separate systems for text and image processing, often requiring custom code and low-level configurations from developers. Maintaining these systems incurs higher costs, complexity, and effort.

Azure AI Search addresses these challenges by integrating images into the same retrieval pipeline as text. With a single multimodal pipeline, you can simplify setup and unlock information that resides in charts, screenshots, infographics, scanned forms, and other complex visuals.

Multimodal search is ideal for [retrieval-augmented generation \(RAG\)](#) scenarios. By interpreting the structural logic of images, multimodal search makes your RAG application or AI agent less likely to overlook important visual details. It also provides your users with detailed answers that can be traced back to their original sources, regardless of the source's modality.

How does multimodal search work?

To simplify the creation of a multimodal pipeline, Azure AI Search offers the [Import data \(new\)](#) wizard in the Azure portal. The wizard helps you configure a data source, define extraction and enrichment settings, and generate a multimodal index that contains text, embedded image references, and vector embeddings. For more information, see [Quickstart: Multimodal search in the Azure portal](#).

The wizard follows these steps to create a multimodal pipeline:

- 1. Extract content:** Choose from the [Document Extraction skill](#), [Document Layout skill](#), or [Azure Content Understanding skill](#) to obtain page text, inline images, and structural metadata. Each skill offers different capabilities for metadata extraction, table handling, and file format support. For detailed comparisons, see [Options for multimodal content extraction](#).
- 2. Chunk text:** The [Text Split skill](#) breaks the extracted text into manageable chunks for use in the remaining pipeline, such as the embedding skill.
- 3. Generate image descriptions:** The [GenAI Prompt skill](#) verbalizes images, producing concise natural-language descriptions for text search and embedding using a large language model (LLM).
- 4. Generate embeddings:** The embedding skill creates vector representations of text and images, enabling similarity and hybrid retrieval. You can call [Azure OpenAI](#), [Microsoft Foundry](#), or [Azure Vision](#) embedding models natively.
Alternatively, you can skip image verbalization and pass the extracted text and images directly to a multimodal embedding model through the [AML skill](#) or [Azure Vision multimodal embeddings skill](#). For more information, see [Options for multimodal content embedding](#).
- 5. Store extracted images:** The [knowledge store](#) contains extracted images that can be returned directly to client applications. When you use the wizard, an image's location is stored directly in the multimodal index, enabling convenient retrieval at query time.

Tip

To see multimodal search in action, plug your wizard-created index into the [multimodal RAG sample application](#) ↗. The sample demonstrates how a RAG application consumes a multimodal index and renders both textual citations and associated image snippets in the response. The sample also showcases the code-based process of data ingestion and indexing.

Options for multimodal content extraction

A multimodal pipeline begins by cracking each source document into chunks of text, inline images, and associated metadata. For this step, Azure AI Search provides three built-in skills:

- Document Extraction skill
- Document Layout skill
- Azure Content Understanding skill

 Expand table

Characteristic	Document Extraction skill	Document Layout skill	Azure Content Understanding skill
Text location metadata extraction (pages and bounding polygons)	No	Yes	Yes
Image location metadata extraction (pages and bounding polygons)	Yes	Yes	Yes
Table extraction and preservation	No	No	Yes (including cross-page tables)
Cross-page semantic units	Not applicable	Single page only	Yes (spans page boundaries)
Location metadata extraction based on file type	PDFs only.	Multiple supported file types according to the Azure Document Intelligence in Foundry Tools layout model .	Multiple supported file types , including PDF, DOCX, XLSX, and PPTX.
Billing for data extraction	Image extraction is billed according to Azure AI Search pricing .	Billed according to Document Layout pricing .	Billed according to Azure Content Understanding pricing .
Built-in chunking	No (use Text Split skill)	Yes (based on paragraph boundaries)	Yes (semantic chunking)
Recommended scenarios	Rapid prototyping or production pipelines where the exact position or detailed	RAG pipelines and agent workflows that need precise page numbers, on-page	Advanced document analysis requiring cross-page table extraction, semantic chunking, or consistent handling across

Characteristic	Document Extraction skill	Document Layout skill	Azure Content Understanding skill
	layout information isn't required.	highlights, or diagram overlays in client apps.	document formats (PDF, DOCX, XLSX, PPTX).

Options for multimodal content embedding

In Azure AI Search, retrieving knowledge from images can follow two complementary paths: image verbalization or direct embeddings. Understanding the distinctions helps you align cost, latency, and answer quality with the needs of your application.

Image verbalization followed by text embeddings

With this method, the [GenAI Prompt skill](#) invokes an LLM during ingestion to create a concise natural-language description of each extracted image, such as "Five-step HR access workflow that begins with manager approval." The description is stored as text and embedded alongside the surrounding document text, which you can then vectorize by calling the [Azure OpenAI](#), [Microsoft Foundry](#), or [Azure Vision](#) embedding models.

Because the image is now expressed in language, Azure AI Search can:

- Interpret the relationships and entities shown in a diagram.
- Supply ready-made captions that an LLM can cite verbatim in a response.
- Return relevant snippets for RAG applications or AI agent scenarios with grounded data.

The added semantic depth entails an LLM call for every image and a marginal increase in indexing time.

Direct multimodal embeddings

A second option is to pass the document-extracted images and text to a multimodal embedding model that produces vector representations in the same vector space. Configuration is straightforward, and no LLM is required at indexing time. Direct embeddings are well suited to visual similarity and "find-me-something-that-looks-like-this" scenarios.

Because the representation is purely mathematical, it doesn't convey why two images are related, and it doesn't offer the LLM ready context for citations or detailed explanations.

Combining both approaches

Many solutions need both encoding paths. Diagrams, flow charts, and other explanation-rich visuals are verbalized so that semantic information is available for RAG and AI agent grounding. Screenshots, product photos, or artwork are embedded directly for efficient similarity search. You can customize your Azure AI Search index and indexer skillset pipeline so it can store the two sets of vectors and retrieve them side by side.

Options for querying multimodal content

If your multimodal pipeline is powered by the GenAI Prompt skill, you can run [hybrid queries](#) over both plain text and verbalized images in your search index. You can also use filters to narrow the search results to specific content types, such as only text or only images.

Although the GenAI Prompt skill supports text-to-vector queries via hybrid search, it doesn't support [image-to-vector queries](#). Only the multimodal embedding models provide the vectorizers that convert images into vectors at query time.

To use images as query inputs for your multimodal index, you must use the [AML skill](#) or [Azure Vision multimodal embeddings skill](#) with an equivalent vectorizer. For more information, see [Configure a vectorizer in a search index](#).

Tutorials and samples

To help you get started with multimodal search in Azure AI Search, here's a collection of content that demonstrates how to create and optimize multimodal indexes using Azure functionality.

 Expand table

Content	Description
Quickstart: Multimodal search in the Azure portal	Create and test a multimodal index in the Azure portal using the wizard and Search Explorer.
Tutorial: Verbalize images using generative AI	Extract text and images, verbalize diagrams, and embed the resulting descriptions and text into a searchable index.
Tutorial: Vectorize images and text	Use a vision-text model to embed both text and images directly, enabling visual-similarity search over scanned PDFs.
Tutorial: Verbalize images from a structured document layout	Apply layout-aware chunking and diagram verbalization, capture location metadata, and store cropped images for precise citations and page highlights.
Tutorial: Vectorize from a	Combine layout-aware chunking with unified embeddings for hybrid

Content	Description
structured document layout	semantic and keyword search that returns exact hit locations.
Sample app: Multimodal RAG GitHub repository ↗	An end-to-end, code-ready RAG application with multimodal capabilities that surfaces both text snippets and image annotations. Ideal for jump-starting enterprise copilots.

Last updated on 11/21/2025

Retrieval Augmented Generation (RAG) in Azure AI Search

10/15/2025

Retrieval Augmented Generation (RAG) is a design pattern that augments the capabilities of a chat completion model like ChatGPT by adding an information retrieval step, incorporating your proprietary enterprise content for answer formulation. For an enterprise solution, it's possible to fully constrain generative AI to your enterprise content.

The decision about which information retrieval system to use is critical because it determines the inputs to the LLM. The information retrieval system should provide:

- Indexing strategies that load and refresh at scale, for all of your content, at the frequency you require.
- Query capabilities and relevance tuning. The system should return *relevant* results, in the short-form formats necessary for meeting the token length requirements of large language model (LLM) inputs. Query turnaround should be as fast as possible.
- Security, global reach, and reliability for both data and operations.
- Integration with embedding models for indexing, and chat models or language understanding models for retrieval.

Azure AI Search is a [proven solution for information retrieval](#) in a RAG architecture. It provides indexing and query capabilities, with the infrastructure and security of the Azure cloud. Through code and other components, you can design a comprehensive RAG solution that includes all of the elements for generative AI over your proprietary content.

You can choose between two approaches for RAG workloads: agentic retrieval, or the original query architecture for classic RAG.

(!) Note

New to copilot and RAG concepts? Watch [Vector search and state of the art retrieval for Generative AI apps](#).

Modern RAG with agentic retrieval

Azure AI Search now provides **agentic retrieval**, a specialized pipeline designed specifically for RAG patterns. This approach uses large language models to intelligently break down complex

user queries into focused subqueries, executes them in parallel, and returns structured responses optimized for chat completion models.

Agentic retrieval represents the evolution from traditional single-query RAG patterns to multi-query intelligent retrieval, providing:

- Context-aware query planning using conversation history
- Parallel execution of multiple focused subqueries
- Structured responses with grounding data, citations, and execution metadata
- Built-in semantic ranking for optimal relevance
- Optional answer synthesis that uses an LLM-formulated answer in the query response.

You need new objects for this pipeline: one or more knowledge sources, a knowledge agent, and the retrieve action that you call from application code, such as a tool that works with your agent.

For new RAG implementations, we recommend starting with [agentic retrieval](#). For existing solutions, consider migrating to take advantage of improved accuracy and context understanding.

Classic RAG pattern for Azure AI Search

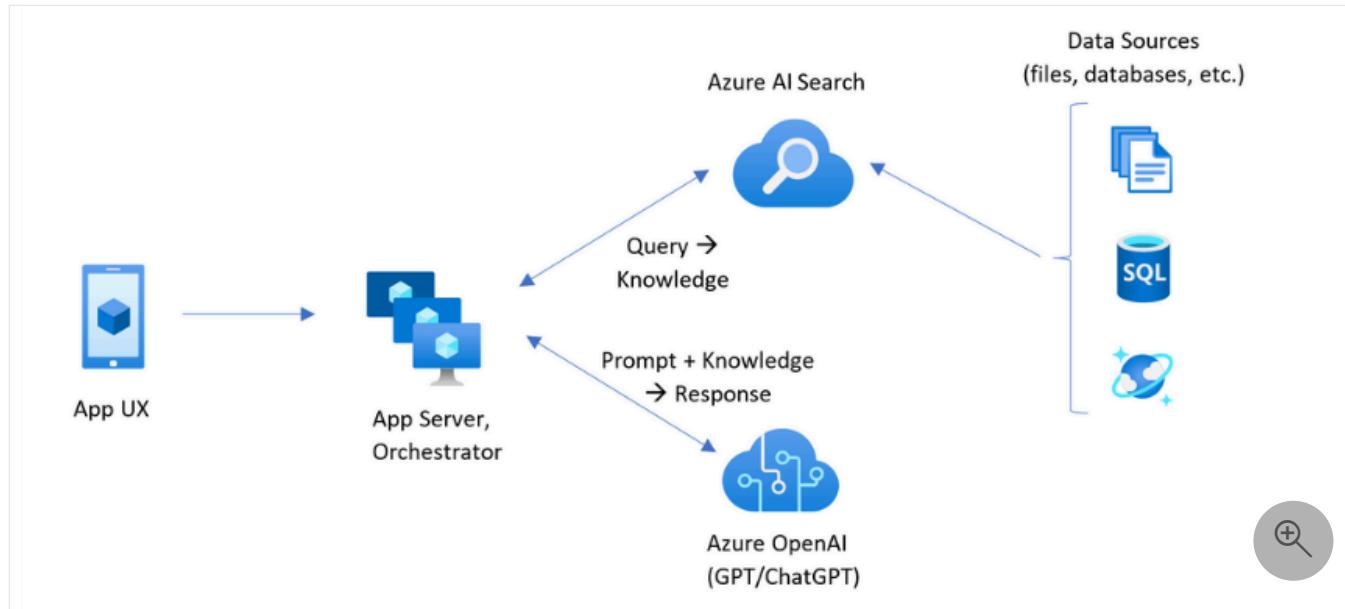
A RAG solution can be implemented on Azure AI Search using the original query execution architecture. With this approach, your application sends a single query request to Azure AI Search, the search engine processes the request, and returns search results to the caller. There's no side trip to an LLM for query planning or answer formulation. There's no query execution details in the response, and citations are built into the response only if you have fields in your index that provide a parent document name or page. This approach is faster and simpler with fewer components. Depending on your application requirements, it could be the best choice.

A high-level summary of classic RAG pattern built on Azure AI Search looks like this:

- Start with a user question or request (prompt).
- Send it as a query request to Azure AI Search to find relevant information.
- Return the top-ranked search results to an LLM.
- Use the natural language understanding and reasoning capabilities of the LLM to generate a response to the initial prompt.

Azure AI Search provides inputs to the LLM prompt, but doesn't train the model. In a traditional RAG pattern, there's no extra training. The LLM is pretrained using public data, but it generates responses that are augmented by information from the retriever, in this case, Azure AI Search.

RAG patterns that include Azure AI Search have the elements indicated in the following illustration.



- App UX (web app) for the user experience
- App server or orchestrator (integration and coordination layer)
- Azure AI Search (information retrieval system)
- Azure OpenAI (LLM for generative AI)

The web app provides the user experience, providing the presentation, context, and user interaction. Questions or prompts from a user start here. Inputs pass through the integration layer, going first to information retrieval to get the search results, but also go to the LLM to set the context and intent.

The app server or orchestrator is the integration code that coordinates the handoffs between information retrieval and the LLM. Common solutions include [Azure Semantic Kernel](#) or [LangChain](#) to coordinate the workflow. [LangChain integrates with Azure AI Search](#), making it easier to include Azure AI Search as a [retriever](#). [Llamaindex](#) and [Semantic Kernel](#) are other options.

The information retrieval system provides the searchable index, query logic, and the payload (query response). The search index can contain vectors or nonvector content. Although most samples and demos include vector fields, it's not a requirement. The query is executed using the existing search engine in Azure AI Search, which can handle keyword (or term) and vector queries. The index is created in advance, based on a schema you define, and loaded with your content that's sourced from files, databases, or storage.

The LLM receives the original prompt, plus the results from Azure AI Search. The LLM analyzes the results and formulates a response. If the LLM is ChatGPT, the user interaction might consist of multiple conversation turns. An Azure solution most likely uses Azure OpenAI, but there's no hard dependency on this specific service.

Searchable content in Azure AI Search

In Azure AI Search, all searchable content is stored in a search index that's hosted on your search service. A search index is designed for fast queries with millisecond response times, so its internal data structures exist to support that objective. To that end, a search index stores *indexed content*, and not whole content files like entire PDFs or images. Internally, the data structures include inverted indexes of [tokenized text](#), vector indexes for embeddings, and unaltered plain text for cases where verbatim matching is required (for example, in filters, fuzzy search, regular expression queries).

When you set up the data for your RAG solution, you use the features that create and load an index in Azure AI Search. An index includes fields that duplicate or represent your source content. An index field might be simple transference (a title or description in a source document becomes a title or description in a search index), or a field might contain the output of an external process, such as vectorization or skill processing that generates a representation or text description of an image.

Since you probably know what kind of content you want to search over, consider the indexing features that are applicable to each content type:

 Expand table

Content type	Indexed as	Features
text	tokens, unaltered text	Indexers can pull plain text from other Azure resources like Azure Storage and Cosmos DB. You can also push any JSON content to an index. To modify text in flight, use analyzers and normalizers to add lexical processing during indexing. Synonym maps are useful if source documents are missing terminology that might be used in a query.
text	vectors ¹	Text can be chunked and vectorized in an indexer pipeline, or handled externally and then indexed as vector fields in your index.
image	tokens, unaltered text ²	Skills for OCR and Image Analysis can process images for text recognition or image characteristics. Skills have an indexer requirement.
image	vectors ¹	Images can be vectorized in an indexer pipeline, or handled externally for a mathematical representation of image content and then indexed as vector fields in your index. You can use Azure AI Vision multimodal or an open source model like OpenAI CLIP to vectorize text and images in the same embedding space.

¹ Azure AI Search provides [integrated data chunking and vectorization](#), but you must take a dependency on indexers and skillsets. If you can't use an indexer, Microsoft's [Semantic Kernel](#)

or other community offerings can help you with a full stack solution. For code samples showing both approaches, see [azure-search-vectors-sample repo ↗](#).

² Image descriptions are converted to searchable text and added to the index. The images themselves are not stored in the index. **Skills** are built-in support for [applied AI](#). For image descriptions and verbalizations, the indexing pipeline makes an internal call to the Azure OpenAI or Azure AI Vision. These skills pass an extracted image for processing, and receive the output as text that's indexed by Azure AI Search. Skills are also used for integrated data chunking and embedding.

Vectors provide the best accommodation for dissimilar content (multiple file formats and languages) because content is expressed universally in mathematic representations. Vectors also support similarity search: matching on the coordinates that are most similar to the vector query. Compared to keyword search (or term search) that matches on tokenized terms, similarity search is more nuanced. It's a better choice if there's ambiguity or interpretation requirements in the content or in queries.

Content retrieval in Azure AI Search

Once your data is in a search index, you use the query capabilities of Azure AI Search to retrieve content.

In a non-RAG pattern, queries make a round trip from a search client. The query is submitted, it executes on a search engine, and the response returned to the client application. The response, or search results, consist exclusively of the verbatim content found in your index.

In a classic RAG pattern, queries and responses are coordinated between the search engine and the LLM. A user's question or query is forwarded to both the search engine and to the LLM as a prompt. The search results come back from the search engine and are redirected to an LLM. The response that makes it back to the user is generative AI, either a summation or answer from the LLM.

In a modern agentic retrieval RAG pattern, queries and responses integrate with LLMs for help with query planning and optional answer formulation. Query inputs can be richer, with chat history as well as the user's question. The LLM determines how to set up subqueries for the best coverage over your indexed content. The response includes not just search results, but the query execution details and source documents. You can optionally include answer formulation, which in other patterns occurs outside of the query pipeline.

Here are the capabilities in Azure AI Search that are used to formulate queries:

Query feature	Purpose	Why use it
Agentic search (preview)	Parallel query execution pipeline of multiple subqueries, returning a response designed for RAG workloads and agent consumer. Queries can be vector or keyword search. Semantic ranking ensures the best results of subquery are included in the final response structure. This is the recommended approach for RAG patterns based on Azure AI Search.	
Keyword search	Query execution over text and nonvector numeric content	Full text search is best for exact matches, rather than similar matches. Full text search queries are ranked using the BM25 algorithm and support relevance tuning through scoring profiles. It also supports filters and facets.
Vector search	Query execution over vector fields for similarity search, where the query string is one or more vectors.	Vectors can represent all types of content, in any language.
Hybrid search	Combines any or all of the above query techniques. Vector and nonvector queries execute in parallel and are returned in a unified result set.	The most significant gains in precision and recall are through hybrid queries.
Filters and facets	Applies to text or numeric (nonvector) fields only. Reduces the search surface area based on inclusion or exclusion criteria.	Adds precision to your queries.

Structure the query response

A query's response provides the input to the LLM, so the quality of your search results is critical to success. Results are a tabular row set. The composition or structure of the results depends on:

- Fields that determine which parts of the index are included in the response.
- Rows that represent a match from index.

Fields appear in search results when the attribute is " retrievable". A field definition in the index schema has attributes, and those determine whether a field is used in a response. Only " retrievable" fields are returned in full text or vector query results. By default all " retrievable" fields are returned, but you can use " select" to specify a subset. Besides " retrievable", there are no restrictions on the field. Fields can be of any length or type. Regarding length, there's no maximum field length limit in Azure AI Search, but there are limits on the [size of an API request](#).

Rows are matches to the query, ranked by relevance, similarity, or both. By default, results are capped at the top 50 matches for full text search or k-nearest-neighbor matches for vector search. You can change the defaults to increase or decrease the limit up to the maximum of 1,000 documents. You can also use top and skip paging parameters to retrieve results as a series of paged results.

Maximize relevance and recall

When you're working with complex processes, a large amount of data, and expectations for millisecond responses, it's critical that each step adds value and improves the quality of the end result. On the information retrieval side, *relevance tuning* is an activity that improves the quality of the results sent to the LLM. Only the most relevant or the most similar matching documents should be included in results.

Here are some tips for maximizing relevance and recall:

- [Hybrid queries](#) that combine keyword (nonvector) search and vector search give you maximum recall when the inputs are the same. In a hybrid query, if you double down on the same input, a text string and its vector equivalent generate parallel queries for keywords and similarity search, returning the most relevant matches from each query type in a unified result set.
- Hybrid queries can also be expansive. You can run similarity search over verbose chunked content, and keyword search over names, all in the same request.
- Relevance tuning is supported through:
 - [Scoring profiles](#) that boost the search score if matches are found in a specific search field or on other criteria.
 - [Semantic ranker](#) that re-ranks an initial results set, using semantic models from Bing to reorder results for a better semantic fit to the original query.
 - Query parameters for fine-tuning. You can [boost the importance of vector queries](#) or [adjust the amount of BM25-ranked results](#) in a hybrid query response. You can also [set minimum thresholds to exclude low scoring results](#) from a vector query.

In comparison and benchmark testing, hybrid queries with text and vector fields, supplemented with semantic ranking, produce the most relevant results.

Integration code and LLMs

A RAG solution that includes Azure AI Search can leverage [built-in data chunking and vectorization capabilities](#), or you can build your own using platforms like Semantic Kernel, LangChain, or LlamaIndex.

We recommend the [Azure OpenAI demo ↗](#) for an example of a full stack RAG chat app with Azure OpenAI and Azure AI Search.

How to get started

There are many ways to get started, including code-first solutions and demos.

For help with choosing between agentic retrieval and classic RAG, try a few quickstarts using your own data to understand the development effort and compare outcomes.

Docs

- [Try this agentic retrieval quickstart](#) to walk through the new and recommended approach for RAG.
- [Try this classic RAG quickstart](#) for a demonstration of query integration with chat models over a search index.
- [Review indexing concepts and strategies](#) to determine how you want to ingest and refresh data. Decide whether to use vector search, keyword search, or hybrid search. The kind of content you need to search over, and the type of queries you want to run, determines index design.
- [Review creating queries](#) to learn more about search request syntax and requirements.

ⓘ Note

Some Azure AI Search features are intended for human interaction and aren't useful in a RAG pattern. Specifically, you can skip features like autocomplete and suggestions. Other features like facets and orderby might be useful, but would be uncommon in a RAG scenario.

See also

- [RAG Experiment Accelerator ↗](#)

- Retrieval Augmented Generation: Streamlining the creation of intelligent natural language processing models ↗
- Azure Cognitive Search and LangChain: A Seamless Integration for Enhanced Vector Search Capabilities ↗

Querying in Azure AI Search

05/29/2025

Azure AI Search supports query constructs for a broad range of scenarios, from free-form text search, to highly specified query patterns, to vector search. All queries execute over a search index that stores searchable content.

Types of queries

 Expand table

Query form	Searchable content	Description
Full text search	Inverted indexes of tokenized terms.	Full text queries iterate over inverted indexes that are structured for fast scans, where a match can be found in potentially any field, within any number of search documents. Text is analyzed and tokenized for full text search.
Vector search	Vector indexes of generated embeddings.	Vector queries iterate over vector fields in a search index.
Hybrid search	All of the above, in a single search index.	Combines text search and vector search in a single query request. Text search works on plain text content in "searchable" and "filterable" fields. Vector search works on content in vector fields.
Agentic retrieval (preview)	All of the above, in a single search index.	This is an alternative retrieval path on Azure AI Search that leverages large language models for query planning. The response is designed for agent consumption, where the agent rather than search app client code coordinates the response delivered to the user.
Others	Plain text and human-readable content.	Raw content, extracted verbatim from source documents, supporting filters and pattern matching queries like geo-spatial search, fuzzy search, and fielded search.

The remainder of this article brings focus to the last category: classic queries that work on plain text and human-readable content, extracted intact from original source, used for filters and other specialized query forms. If you're creating a traditional search application that isn't using AI, this section explains the query methods that you can implement in your client code.

Autocomplete and suggested queries

Autocomplete or suggested results are alternatives to `search` that fire successive query requests based on partial string inputs (after each character) in a search-as-you-type experience. You can use `autocomplete` and `suggestions` parameter together or separately, as described in [this walkthrough](#), but you can't use them with `search`. Both completed terms and suggested queries are derived from index contents. The engine never returns a string or suggestion that is nonexistent in your index. For more information, see [Autocomplete \(REST API\)](#) and [Suggestions \(REST API\)](#).

Filter search

Filters are widely used in apps that are based on Azure AI Search. On application pages, filters are often visualized as facets in link navigation structures for user-directed filtering. Filters are also used internally to expose slices of indexed content. For example, you might initialize a search page using a filter on a product category, or a language if an index contains fields in both English and French.

You might also need filters to invoke a specialized query form, as described in the following table. You can use a filter with an unspecified search (`search=*`) or with a query string that includes terms, phrases, operators, and patterns.

[] [Expand table](#)

Filter scenario	Description
Range filters	In Azure AI Search, range queries are built using the <code>filter</code> parameter. For more information and examples, see Range filter example .
Faceted navigation	In faceted navigation tree, users can select facets. When backed by filters, search results narrow on each click. Each facet is backed by a filter that excludes documents that no longer match the criteria provided by the facet.

! Note

Text that's used in a filter expression is not analyzed during query processing. The text input is presumed to be a verbatim case-sensitive character pattern that either succeeds or fails on the match. Filter expressions are constructed using [OData syntax](#) and passed in a `filter` parameter in all *filterable* fields in your index. For more information, see [Filters in Azure AI Search](#).

Geospatial search

Geospatial search matches on a location's latitude and longitude coordinates for "find near me" or map-based search experience. In Azure AI Search, you can implement geospatial search by following these steps:

- Define a filterable field of one of these types: [Edm.GeographyPoint](#), [Collection\(Edm.GeographyPoint\)](#), [Edm.GeographyPolygon](#).
- Verify the incoming documents include the appropriate coordinates.
- After indexing is complete, build a query that uses a filter and a [geo-spatial function](#).

Geospatial search uses kilometers for distance. Coordinates are specified in this format:

(longitude, latitude).

Here's an example of a filter for geospatial search. This filter finds other `Location` fields in the search index that have coordinates within a 300-kilometer radius of the geography point (in this example, Washington D.C.). It returns address information in the result, and includes an optional `facets` clause for self-navigation based on location.

HTTP

```
POST https://{{searchServiceName}}.search.windows.net/indexes/hotels-vector-quickstart/docs/search?api-version=2024-07-01
{
    "count": true,
    "search": "*",
    "filter": "geo.distance(Location, geography'POINT(-77.03241 38.90166)') le 300",
    "facets": [ "Address/StateProvince"],
    "select": "HotelId, HotelName, Address/StreetAddress, Address/City, Address/StateProvince",
    "top": 7
}
```

For more information and examples, see [Geospatial search example](#).

Document look-up

In contrast with the previously described query forms, this one retrieves a single [search document by ID](#), with no corresponding index search or scan. Only the one document is requested and returned. When a user selects an item in search results, retrieving the document and populating a details page with fields is a typical response, and a document look-up is the operation that supports it.

Advanced search: fuzzy, wildcard, proximity, regex

An advanced query form depends on the Full Lucene parser and operators that trigger a specific query behavior.

[] [Expand table](#)

Query type	Usage	Examples and more information
Fielded search	<code>search</code> parameter, <code>queryType=full</code>	Build a composite query expression targeting a single field. Fielded search example
fuzzy search	<code>search</code> parameter, <code>queryType=full</code>	Matches on terms having a similar construction or spelling. Fuzzy search example
proximity search	<code>search</code> parameter, <code>queryType=full</code>	Finds terms that are near each other in a document. Proximity search example
term boosting	<code>search</code> parameter, <code>queryType=full</code>	Ranks a document higher if it contains the boosted term, relative to others that don't. Term boosting example
regular expression search	<code>search</code> parameter, <code>queryType=full</code>	Matches based on the contents of a regular expression. Regular expression example
wildcard or prefix search	<code>search</code> parameter with <code>*~</code> or <code>?</code> , <code>queryType=full</code>	Matches based on a prefix and tilde (~) or single character (?). Wildcard search example

Next steps

For a closer look at query implementation, review the examples for each syntax. If you're new to full text search, a closer look at what the query engine does might be an equally good choice.

- [Simple query examples](#)
- [Lucene syntax query examples for building advanced queries](#)
- [How full text search works in Azure AI Search](#)

Relevance in Azure AI Search

The true measure of relevance is *how well* a retrieved set of results meets your customer and user information needs. In this article, learn about:

- The main strategies for producing relevant results in Azure AI Search
- The mechanics of how relevance is measured
- The actions you can take to improve relevance

Strategies for highly relevant results

In Azure AI Search, two main strategies have emerged as the best approaches for producing highly relevant results.

- Hybrid search with semantic reranker
- Agentic retrieval (preview) with LLM-assisted query planning and answer formulation

[Hybrid search](#) delivers on relevance by combining the precision of keyword queries and the semantic similarity of vector queries in a search request targeting a single index. Keyword search operates over a verbatim query. Vector search runs an identical query using a vectorized version of the same string. The queries execute in parallel, looking for precise and semantically similar matches. Results are merged, ranked, and then rescored using a semantic ranker that promotes the most relevant matches. Using keyword and vector search *together* offsets the weaknesses of each approach as a standalone solution. Semantic reranker is an extra component that contributes to a better outcome.

[Agentic retrieval \(preview\)](#) delivers on relevance through smart integration with LLMs and a knowledge base that defines an entire search domain. The LLM can analyze and transform queries for more effective retrieval. It can decompose complex questions into targeted subqueries, refine vague requests, or generalize narrow ones for broader scope. In a typical agentic retrieval workload, the LLM answers the question using its reasoning power, context from chat history, and retrieval instructions to extract the very best content and use it to best advantage. This combination of LLM-assisted query planning, multi-source knowledge base search, and LLM reasoning is how agentic retrieval returns highly relevant results.

Relevance also depends on having grounding data of sufficient quantity and quality. In agentic retrieval, you can list multiple knowledge sources to expand the scope of what's searchable and provide logic for selecting specific ones.

How relevance is measured

Regardless of how content is retrieved, the relevance of any given result is determined by a ranking algorithm that evaluates the strength of a match based on how closely the query corresponds to content in the search corpus. When a match is found, an algorithm assigns a score, and results are ranked by that score and the topmost results are returned in the response.

Ranking occurs whenever the query request is for agentic retrieval and classic search for keyword, vector, and hybrid queries. It doesn't occur if the query invokes strict pattern matching, such as a filter-only query or a specialized query form like autocomplete, suggestions, geospatial search, fuzzy search, or regular expression search. A uniform search score of 1.0 indicates the absence of a ranking algorithm.

Levels of ranking

The query engine in Azure AI Search supports a multi-level approach to ranking search results, where there's a built-in ranking modality for each query type, plus extra ranking capabilities for extended relevance tuning.

This section describes the levels of scoring operations. For an illustration of how they work together, see the [diagram](#) in this article. A [comparison of all search score types and ranges](#) is also provided in this article.

[] Expand table

Level	Description
Level 1 (L1)	<p>Initial search score (<code>@search.score</code>).</p> <p>For text queries matching on tokenized strings, results are always initially ranked using the BM25 ranking algorithm.</p> <p>For vector queries, results are ranked using either Hierarchical Navigable Small World (HNSW) or exhaustive K-nearest neighbor (KNN). Image search or multimodal searches are based on vector queries and scored using the L1 vector ranking algorithms.</p>
Fused L1	<p>Scoring from multiple queries using the Reciprocal Ranking Fusion (RRF) algorithm. RRF is used for hybrid queries that include text and vector components. RRF is also used when multiple vector queries execute in parallel. A search score from RRF is reflected in <code>@search.score</code> over a different range.</p>
Level 2 (L2)	<p>Semantic ranking score (@search.reRankerScore) applies machine reading comprehension to the textual content retrieved by L1 ranking, rescoring the L1 results to better match the semantic intent of the query. L2 reranks L1 results because doing so saves time and money; it would be prohibitive to use semantic ranking as an L1 ranking system. Semantic ranking is a premium feature that bills for usage of the semantic ranking models. It's optional for text queries and vector queries that contain text, but required for agentic retrieval (preview).</p>

Level	Description
	Although agentic retrieval sends multiple queries to the query engine, the ranking algorithm for agentic retrieval is the semantic ranker.
Level 3 (L3)	Applies to agentic retrieval (preview) and a medium retrieval reasoning effort. L3 ranking refers to <i>iterative search</i> and it's invoked when the agentic retrieval engine and LLM agree that a second query pass is needed to return a more relevant result. For more information, see Medium retrieval and iterative search .

Relevance tuning

Relevance tuning is a technique for boosting search scores based on extra criteria such as weighted fields, freshness, or proximity. In Azure AI Search, relevance tuning options vary based on query type:

- For textual and numeric (nonvector) content in keyword or hybrid search, you can tune relevance through [scoring profiles](#) or invoking the [semantic ranker](#).
- For vector content in a hybrid query, you can [weight a vector field](#) to boost the importance of the vector component relative to the text component of the hybrid query.
- For pure vector queries, you can experiment between Hierarchical Navigable Small World (HNSW) and exhaustive K-nearest neighbors (KNN) to see if one algorithm outperforms the other for your scenario. HNSW graphing with an exhaustive KNN override at query time is the most flexible approach for comparison testing. You can also experiment with various embedding models to see which ones produce higher quality results. Finally, remember that a hybrid query or a vector query on documents that include nonvector fields are in-scope for relevance tuning, so it's just the vector fields themselves that can't participate in a relevance tuning effort.

Custom boosting logic using scoring profiles

[Scoring profiles](#) are an optional feature for boosting scores based on extra user-defined criteria. Criteria can include weighted fields where a match found in a specific field is given more weight than the same match found in a different field. Criteria can also be defined through functions that boost by freshness, proximity, magnitude, or range. There's no extra costs associated with scoring profiles. To use a scoring profile, you define it in an index and then specify it on a query.

Scoring logic applies to text and numeric nonvector content. You can use scoring profiles with:

- [Text \(keyword\) search](#)

- Pure vector queries
- Hybrid queries, where text and vector subqueries execute in parallel
- Semantically ranked queries

For standalone text queries, scoring profiles identify the top 1,000 matches in a [BM25-ranked search](#), with the top 50 matches returned in the response.

For pure vectors, the query is vector-only, but if the [k-matching documents](#) include nonvector fields with human-readable content, a scoring profile is applied to nonvector fields in `k` documents.

For the text component of a hybrid query, scoring profiles identify the top 1,000 matches in a BM25-ranked search. However, once those 1,000 results are identified, they're restored to their original BM25 order so that they can be rescored alongside vectors results in the final [Reciprocal Ranking Function \(RRF\)](#) ordering, where the scoring profile (identified as "final document boosting adjustment" in the illustration) is applied to the merged results, along with [vector weighting](#), and [semantic ranking](#) as the last step.

For semantically ranked queries (not shown in the diagram), assuming you use the latest preview REST API or a preview Azure SDK package, scoring profiles can be applied over an L2 ranked result set, generating a new `@search.rerankerBoostedScore` that determines the final ranking.

Types of search scores

Scored results are indicated for each match in the query response. This table lists all of the search scores with an associated range. Range varies by algorithm.

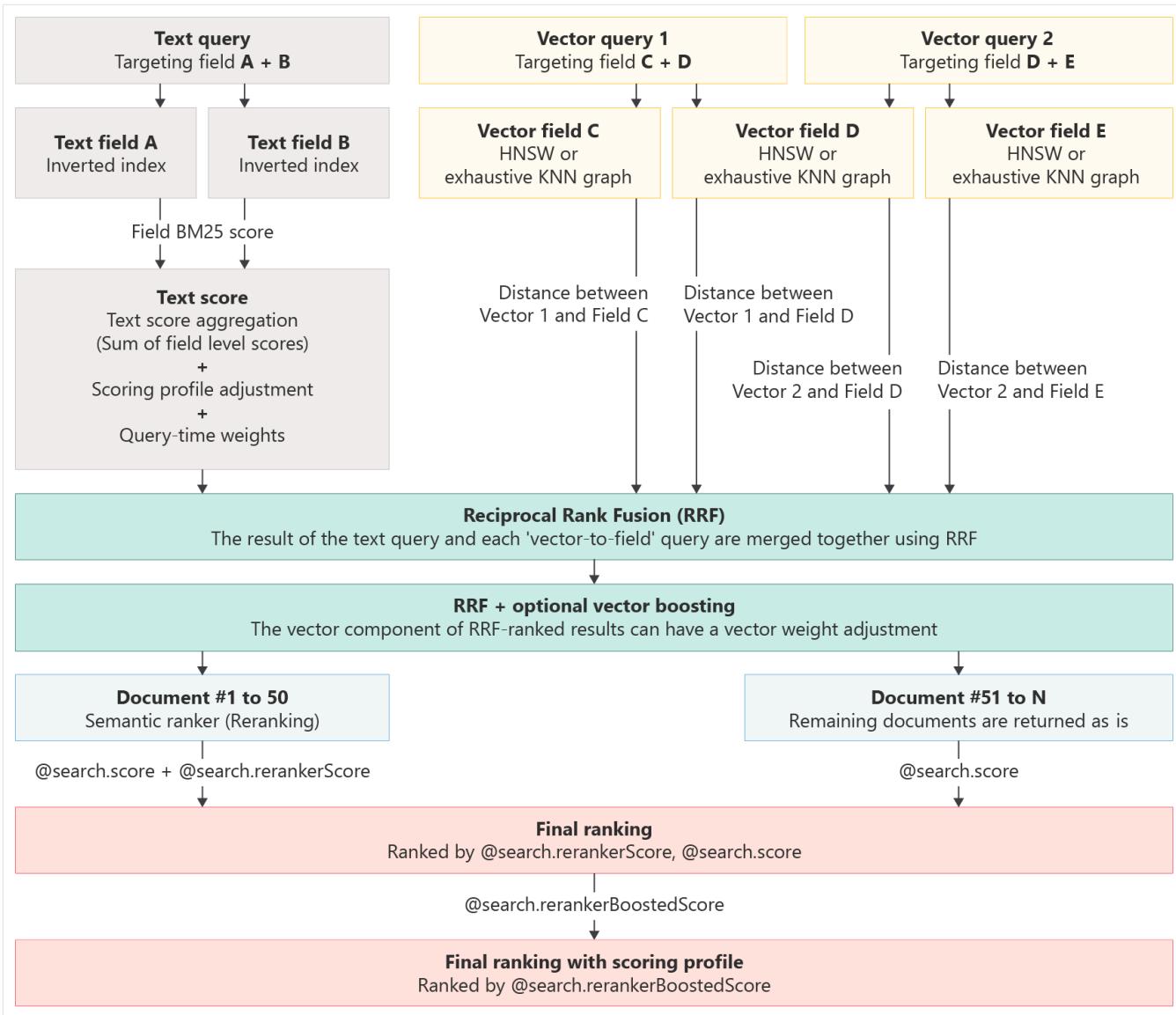
 [Expand table](#)

Score	Range	Algorithm
<code>@search.score</code>	0 through unlimited	BM25 ranking algorithm for text search
<code>@search.score</code>	0.333 - 1.00	HNSW or exhaustive KNN algorithm for vector search
<code>@search.score</code>	0 through an upper limit determined by the number of queries	RRF algorithm
<code>@search.rerankerScore</code>	0.00 - 4.00	Semantic ranking algorithm for L2 ranking

Score	Range	Algorithm
<code>@search.rerankerBoostedScore</code>	0 through unlimited	Semantic ranking with scoring profile boosting (scores can be significantly higher than 4)

Diagram of ranking algorithms

The following diagram illustrates how the ranking algorithms work together.



! Note

If you use semantic ranking, the `rankingOrder` property determines whether results are the semantically scored results (`@search.rerankerScore`) or the boosted scores (`@search.rerankerBoostedScore`) that are created after the scoring profile is applied.

Example query inclusive of all ranking algorithms

A query that generates the previous workflow might look like the following example. This hybrid semantic query is scored using RRF (based on L1 scores for text and vectors), and semantic ranking.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-11-01-preview

{
  "search": "cloud formation over water",
  "count": true,
  "vectorQueries": [
    {
      "kind": "text",
      "text": "cloud formation over water",
      "fields": "text_vector,image_vector"
    }
  ],
  "queryType": "semantic",
  "semanticConfiguration": "my-semantic-configuration",
  "select": "title,chunk",
  "top": 5
}
```

A response for the previous query includes the original RRF `@search.core` and the `@search.rerankerScore`.

JSON

```
"value": [
  {
    "@search.score": 0.03177805617451668,
    "@search.rerankerScore": 2.6919238567352295,
    "chunk":
"A\nT\nnM\nnO\nn\nnS\nnP\nn\nnH\nnE\nn\nnR\nnE\nnA\nn\nnR\nnT\nn\nnH\nn\nn32\nn\nFraming an
Iceberg\nSouth Atlantic Ocean\nIn June 2016, the Suomi NPP satellite captured this
image of various cloud formations in the South Atlantic Ocean. Note how low
\n\nstratus clouds framed a hole over iceberg A-56 as it drifted across the sea.
\n\nThe exact reason for the hole in the clouds is somewhat of a mystery. It could
have formed by chance, although imagery from the \n\ndays before and after this date
suggest something else was at work. It could be that the relatively unobstructed
path of the clouds \n\nover the ocean surface was interrupted by thermal instability
created by the iceberg. In other words, if an obstacle is big enough, \n\nit can
divert the low-level atmospheric flow of air around it, a phenomenon often caused by
islands.",
    "title": "page-39.pdf",
  },
  {
```

"@search.score": 0.030621785670518875,
 "@search.rerankerScore": 2.557225465774536,
 "chunk":
 "A\nT\nnM\nnO\nnS\nnP\nn\nH\nnE\nn\nR\nnE\nnA\nn\nR\nnT\nn\nH\nn\n24\nn\nMaking Tracks\nPacific Ocean\nn\nShips steaming across the Pacific Ocean left this cluster of bright cloud trails lingering in the atmosphere in February 2012. The \n\nnarrow clouds, known as ship tracks, form when water vapor condenses around tiny particles of pollution from ship exhaust. The \n\nncrisscrossing clouds off the coast of California stretched for many hundreds of kilometers from end to end. The narrow ends of the \n\nnclouds are youngest, while the broader, wavier ends are older.\n\nSome of the pollution particles generated by ships (especially sulfates) are soluble in water and can serve as the seeds around which \n\nncloud droplets form. Clouds infused with ship exhaust have more and smaller droplets than unpolluted clouds. As a result, light \n\nnhitting the ship tracks scatters in many directions, often making them appear brighter than other types of marine clouds, which are \n\nnusually seeded by larger, naturally occurring particles like sea salt.",
 "title": "page-31.pdf",
 },
 {
 "@search.score": 0.013698630034923553,
 "@search.rerankerScore": 2.515575408935547,
 "chunk":
 "A\nT\nnM\nnO\nnS\nnP\nn\nH\nnE\nn\nR\nnE\nnA\nn\nR\nnT\nn\nH\nn\n16\nn\nRiding the Waves\nMauritania\nn\nYou cannot see it directly, but air masses from Africa and the Atlantic Ocean are colliding in this Landsat 8 image from August 2016. \n\n\nThe collision off the coast of Mauritania produces a wave structure in the atmosphere. \n\n\nCalled an undular bore or solitary wave, this cloud formation was created by the interaction between cool, dry air coming off the \n\nncontinent and running into warm, moist air over the ocean. The winds blowing out from the land push a wave of air ahead like a \n\nnbow wave moving ahead of a boat. \n\n\nParts of these waves are favorable for cloud formation, while other parts are not. The dust blowing out from Africa appears to be \n\nnriding these waves. Dust has been known to affect cloud growth, but it probably has little to do with the cloud pattern observed here.",
 "title": "page-23.pdf",
 },
 {
 "@search.score": 0.028949543833732605,
 "@search.rerankerScore": 2.4990925788879395,
 "chunk":
 "A\nT\nnM\nnO\nnS\nnP\nn\nH\nnE\nn\nR\nnE\nnA\nn\nR\nnT\nn\nH\nn\n14\nn\nBering Streets\nArctic Ocean\nn\nWinds from the northeast pushed sea ice southward and formed cloud streets—parallel rows of clouds—over the Bering Strait in \n\n\nJanuary 2010. The easternmost reaches of Russia, blanketed in snow and ice, appear in the upper left. To the east, sea ice spans \n\n\nthe Bering Strait. Along the southern edge of the ice, wavy tendrils of newly formed, thin sea ice predominate.\n\n\nThe cloud streets run in the direction of the northerly wind that helps form them. When wind blows out from a cold surface like sea \n\n\nice over the warmer, moister air near the open ocean, cylinders of spinning air may develop. Clouds form along the upward cycle in \n\n\nthe cylinders, where air is rising, and skies remain clear along the downward cycle, where air is falling. The cloud streets run toward \n\n\nthe southwest in this image from the Terra satellite.",
 "title": "page-21.pdf",
 },
 {

```
    "@search.score": 0.027637723833322525,  
    "@search.rerankerScore": 2.4686081409454346,  
    "chunk":  
      "A\nT\nnM\nnO\nn\nnS\nnP\nn\nH\nnE\nn\nR\nnE\nnA\nn\nR\nnT\nn\nnH\nn\nn38\nn\nLofted Over  
Land\nMadagascar\nnAlong the muddy Mania River, midday clouds form over the  
forested land but not the water. In the tropical rainforests of Madagascar,  
\n\nthere is ample moisture for cloud formation. Sunlight heats the land all day,  
warming that moist air and causing it to rise high into the \n\natmosphere until it  
cools and condenses into water droplets. Clouds generally form where air is  
ascending (over land in this case), \n\nbut not where it is descending (over the  
river). Landsat 8 acquired this image in January 2015.",  
    "title": "page-45.pdf",  
  }  
]
```

Last updated on 12/08/2025

Reliability in Azure AI Search

Azure AI Search is a scalable search infrastructure that indexes heterogeneous content and enables retrieval through APIs, applications, and AI agents. It's suitable for enterprise search scenarios and AI-powered customer experiences that require dynamic content generation through chat completion models. As an Azure service, AI Search provides a range of capabilities to support your reliability requirements.

When you use Azure, [reliability is a shared responsibility](#). Microsoft provides a range of capabilities to support resiliency and recovery. You're responsible for understanding how those capabilities work within all of the services you use, and selecting the capabilities you need to meet your business objectives and uptime goals.

This article describes how to make Azure AI Search resilient to a variety of potential outages and problems, including transient faults, availability zone outages, region outages, and service maintenance. It also describes how you can use backups to recover from other types of problems, and highlights some key information about the Azure AI Search service level agreement (SLA).

Production deployment recommendations for reliability

For production workloads, we recommend that you:

- ✓ Use a [billable tier](#) that has at least [two replicas](#). This configuration makes your search service more resilient to transient faults and maintenance operations. It also meets the [service-level agreement \(SLA\)](#) for AI Search. The SLA requires two replicas for read-only workloads and three or more replicas for read-write workloads.
- ✓ Don't use the Free tier for production use. AI Search doesn't provide an SLA for the Free tier, which is limited to one replica.

Reliability architecture overview

When you use AI Search, you create a *search service*. Each search service supports many *search indexes* that store your searchable content.

AI Search isn't designed as a primary data store. Instead, you use *indexers* to connect your search service to external data sources. An indexer crawls the source data, invokes *skills* that perform processing and enrichment, and populates your index with the skill outputs.

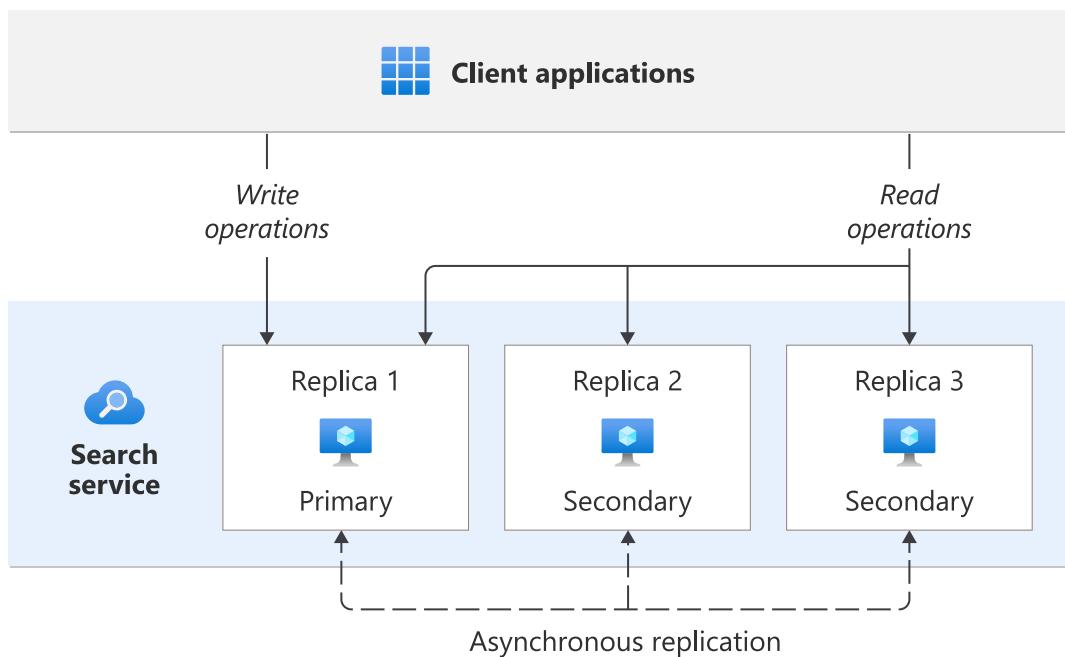
You also configure the number of *replicas* for your service. In AI Search, a replica is a copy of your service's search engine. You can think of a replica as representing a single virtual machine (VM). Each search service can have between 1 and 12 replicas.

The addition of multiple replicas allows AI Search to:

- Increase the availability of your search service.
- Perform maintenance on one replica while queries continue to run on other replicas.
- Handle higher indexing and query workloads.
- Improve resiliency by attempting to provision replicas in different availability zones, if your region supports them.

AI Search automatically assigns one replica to be the *primary replica*. All write operations are performed against that replica. The other replicas are used for read operations.

The following diagram illustrates how a search service with three replicas might be spread across three availability zones:



You can also configure the number of *partitions*, which represent the storage that the search indexes use.

It's important to understand the impact of adding replicas and partitions because they each affect read and write performance in different ways. For more information about replicas and partitions, see [Estimate and manage capacity of a search service](#).

Resilience to transient faults

Transient faults are short, intermittent failures in components. They occur frequently in a distributed environment like the cloud, and they're a normal part of operations. Transient faults correct themselves after a short period of time. It's important that your applications can handle transient faults, usually by retrying affected requests.

All cloud-hosted applications should follow the Azure transient fault handling guidance when they communicate with any cloud-hosted APIs, databases, and other components. For more information, see [Recommendations for handling transient faults](#).

AI Search indexers have built-in transient fault handling. If a data source is briefly unavailable, the indexer is designed to recover and retry. It uses change tracking to resume indexing from the last successfully indexed document.

Search services might experience transient faults during standard, unscheduled maintenance operations. Azure AI Search doesn't provide advance notification or allow scheduling of maintenance at specific times. Although every effort is made to minimize downtime, even for single-replica services, brief interruptions can still occur. To improve resiliency against these transient faults, we recommend that you use two or more replicas.

If you build any applications that interact with AI Search, they should handle transient faults. Use a retry strategy with exponential backoffs for both read and write operations.

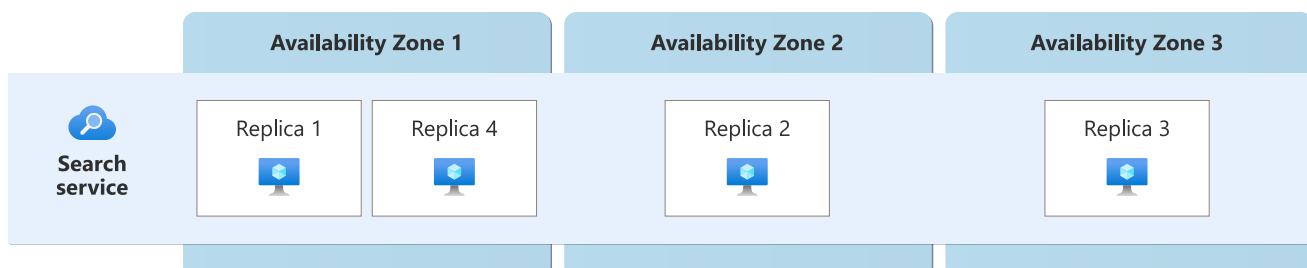
Resilience to availability zone failures

[Availability zones](#) are physically separate groups of datacenters within each Azure region. When one zone fails, services can fail over to one of the remaining zones.

AI Search is zone redundant, which means that your replicas are distributed across multiple availability zones within the search service region.

When you add two or more replicas to your service, AI Search attempts to place each replica in a different availability zone. For services that have more replicas than available zones, replicas are distributed across zones as evenly as possible.

The following diagram illustrates how an example search service with four replicas might be deployed across three availability zones:



Important

AI Search doesn't guarantee the exact placement of replicas. Placement is subject to capacity constraints, scaling operations, and other factors.

Requirements

Zone redundancy is automatically enabled when your search service meets all of the following criteria:

- **Region support:** Support for availability zones depends on infrastructure and storage. For a list of supported regions, see [Choose a region for AI Search](#).
- **Tier:** Your service must be on the [Basic tier or higher](#)
- **Number of replicas:** Your service must have [at least two replicas](#)

Note

AI Search attempts to distribute replicas across multiple zones when you have two or more replicas. However, for read-write workloads, you should use three or more replicas so that you receive the highest possible availability SLA.

Instance distribution across zones

AI Search attempts to place replicas across different availability zones. However, there are occasionally situations where all of the replicas of a search service might be placed into the same availability zone. This situation can happen when replicas are removed from your service, such as when you *scale in* by configuring your service to use fewer replicas. Replica removal doesn't trigger the remaining replicas to rebalance across the availability zones.

To reduce the likelihood of all of your replicas being placed into a single availability zone, you can manually trigger a scale-out operation immediately after a scale-in operation. For example, suppose that your search service has 10 replicas and you want to scale in to 7 replicas. Instead of performing a single scale operation, you can temporarily scale to 6 instances and then immediately scale to 7 instances to trigger zone rebalancing.

Cost

Each search service starts with one replica. Zone redundancy requires two or more replicas, which increases the cost to run the service. To understand the billing implications of replicas, use the [pricing calculator](#).

Configure availability zone support

If your search service meets the [requirements for zone redundancy](#), no extra configuration is necessary. Whenever possible, AI Search attempts to place your replicas in different availability zones.

Capacity planning and management

To prepare for availability zone failure, consider *overprovisioning* the number of replicas. Overprovisioning allows the search service to tolerate some capacity loss and continue to function without degraded performance. Adding replicas during an outage is challenging, so overprovisioning helps ensure that your search service can handle normal request volumes, even with reduced capacity. For more information, see [Manage capacity by overprovisioning](#).

Behavior when all zones are healthy

This section describes what to expect when search services are configured for zone redundancy and all availability zones are operational.

- **Traffic routing between zones:** AI Search performs automatic load balancing of all queries and writes across all of the available replicas. AI Search can send read operations to any replica in any availability zone. It sends write operations to a single primary replica that the AI Search service selects.
- **Data replication between zones:** Changes in data are automatically replicated between replicas across availability zones. Replication occurs asynchronously, which means that writes are committed to one primary replica before they're replicated to other replicas.

Behavior during a zone failure

This section describes what to expect when search services are configured for zone redundancy and an availability zone outage occurs.

- **Detection and response:** AI Search is responsible for detecting a failure in an availability zone. You don't need to do anything to initiate a zone failover.
- **Notification:** Microsoft doesn't automatically notify you when a zone is down. However:

- You can use [Azure Resource Health](#) to monitor for the health of an individual resource, and you can set up [Resource Health alerts](#) to notify you of problems.
- You can use [Azure Service Health](#) to understand the overall health of the service, including any zone failures, and you can set up [Service Health alerts](#) to notify you of problems.
- **Active requests:** Requests that replicas process in the failed zone are terminated. Clients should retry the requests by following the guidance for [handling transient faults](#).
- **Expected data loss:** If the affected availability zone only contains read replicas, no data loss is expected.

If the primary replica is lost because it was in the affected zone, then any write operations that haven't yet been replicated might be lost.

- **Expected downtime:** In most situations, a zone failure isn't expected to cause downtime to your search service for read operations because read replicas in other availability zones continue to serve requests.

If the primary replica is lost because it was in the affected zone, AI Search automatically promotes another replica to become the new primary so that write operations can resume. It typically takes a few seconds for the replica promotion to occur. During this time, write operations might not succeed. Ensure that your applications are prepared by following [transient fault handling guidance](#).

However, there are some unlikely situations where all of your search service's replicas might be in a single availability zone. In this scenario, you might experience downtime until the zone recovers. For more information, and to understand a workaround, see [Instance distribution](#).

- **Traffic rerouting:** When a zone fails, AI Search detects the failure and routes requests to active replicas in the surviving zones. If the primary replica is lost, another replica is promoted to be the new primary.

Zone recovery

When the availability zone recovers, AI Search automatically restores normal operations and begins routing traffic to available replicas across all zones, including the recovered zone.

Test for zone failures

AI Search manages traffic routing for zone-redundant services. You don't need to initiate or validate any zone failure processes.

Resilience to region-wide failures

AI Search is a single-region service. If the region becomes unavailable, your search service also becomes unavailable.

Custom multi-region solutions for resiliency

You can optionally deploy multiple AI Search services in different regions. You're responsible for deploying and configuring separate services in each region. If you create an identical deployment in a secondary Azure region that uses a multi-region architecture, your application becomes less susceptible to a single-region disaster.

When you follow this approach, you must synchronize indexes across regions to recover the last application state. You must also configure load balancing and failover policies.

To optimize the performance of your overall solution, look for opportunities to perform indexing on read-only replicas of your data sources. For example, some indexers support reading from a geo-distributed data source's read replicas.

For more information, see [Multi-region deployments in Azure AI Search](#).

Backup and restore

Because AI Search isn't a primary data storage solution, it doesn't provide self-service backup and restore options. However, you can use the `index-backup-restore` sample for [.NET](#) or [Python](#) to back up your index definition and its documents to a series of JSON files, which are then used to restore the index.

However, if you accidentally delete the index and don't have a backup, you can [rebuild the index](#). Rebuilding involves recreating the index on your search service and then reloading it by retrieving data from your primary data store.

Service-level agreement

The service-level agreement (SLA) for Azure services describes the expected availability of each service and the conditions that your solution must meet to achieve that availability expectation. For more information, see [SLAs for online services](#).

In AI Search, the availability SLA applies to search services that:

- Are configured to use a [billable tier](#).
- Have at least two [replicas](#) for read-only workloads (queries).
- Have at least three replicas for read-write workloads (queries and indexing).

Related content

- [Reliability in Azure](#)
- [Service limits in AI Search](#)
- [Choose a region for AI Search](#)
- [Choose a pricing tier for AI Search](#)
- [Plan or add capacity in AI Search](#)

 **Note:** The author created this article with assistance from AI. [Learn more](#)

Last updated on 11/03/2025

Multi-region deployments in Azure AI Search

08/11/2025

Although Azure AI Search is a single-region service, you can achieve higher availability and resiliency by deploying multiple search services with identical configurations and content across multiple regions.

This article describes the components of a multi-region solution, which relies on your custom script or code to handle failover if a service becomes unavailable.

For more information about the reliability features of Azure AI Search, including intra-regional resiliency via availability zones, see [Reliability in Azure AI Search](#).

Why use multiple regions?

If you need two or more search services, creating them in different regions can meet the following operational requirements:

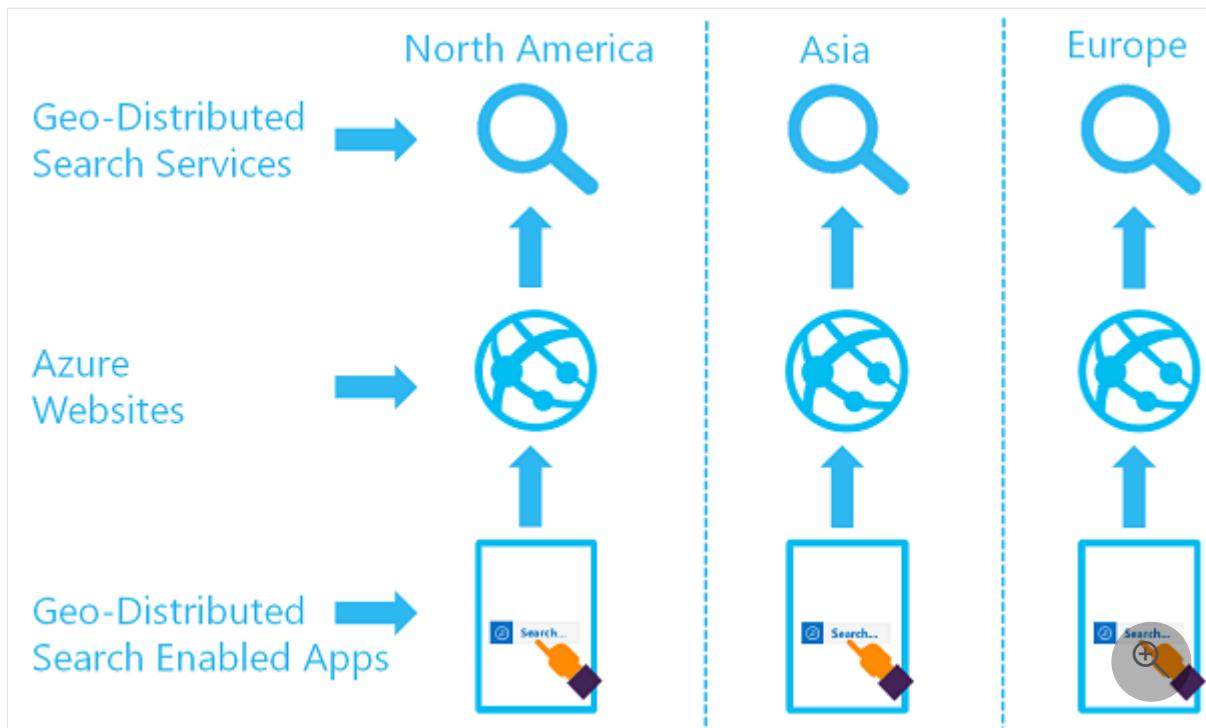
- **Resiliency to region outages.** If there's an outage, Azure AI Search doesn't provide instant failover to another region.
- **Fast performance for a globally distributed application.** If indexing and query requests come from around the world, users who are closest to the host data center experience faster performance. Creating more services in regions with close proximity to these users can equalize performance for everyone.

Multi-region architecture

In a multi-region setup, two or more search services are located in different regions and have synchronized indexes. Users are automatically routed to the service with the lowest latency.

Azure AI Search doesn't provide an automated method of index replication across regions. However, you can synchronize data using [push or pull model indexing](#), both of which are described in the following section. You can also add Azure Traffic Manager or another load balancer for [request redirection](#).

The following diagram illustrates a geo-distributed set of search services:



Tip

For a complete implementation, see the [Bicep sample](#) on GitHub. The sample deploys a fully configured, multi-region search solution that can be modified to your regions and indexing strategies.

Data synchronization

To synchronize two or more distinct search services, you can either:

- Push content into an index using [Documents - Index \(REST API\)](#) or an equivalent API in the Azure SDKs.
- Pull content into an index using an [indexer](#).

Push APIs

If you use the REST APIs to [push content into your index](#), you can synchronize multiple search services by sending updates to each service whenever changes occur. Ensure that your code handles cases in which an update fails for one service but succeeds for other services.

Data residency

When you create multiple search services in different regions, your content is stored in the region you chose for each service.

Azure AI Search doesn't store data outside of your specified region without your authorization. Authorization is implicit when you use features that write to Azure Storage, for which you provide a storage account in your preferred region. These features include:

- [Enrichment cache](#)
- [Debug sessions](#)
- [Knowledge store](#)

If your search service and storage account are in the same region, network traffic uses private IP addresses over the Microsoft backbone network, so you can't configure IP firewalls or private endpoints for network security. As an alternative, use the [trusted service exception](#).

Request failover and redirection

For redundancy at the request level, Azure provides several [load-balancing options](#):

Azure Application Gateway

Use [Azure Application Gateway](#) to load balance between servers in a region at the application layer.

By default, service endpoints are accessed through a public internet connection. Use Application Gateway if you set up a private endpoint for client connections that originate from within a virtual network.

As you evaluate these load-balancing options, consider the following points:

- Azure AI Search is a backend service that accepts indexing and query requests from a client.
- By default, service endpoints are accessed through a public internet connection. We recommend [Azure Application Gateway](#) for private endpoints that originate from within a virtual network.
- Azure AI Search accepts requests addressed to the `<your-search-service-name>.search.windows.net` endpoint. If you reach the same endpoint using a different DNS name in the host header, such as a CNAME, the request is rejected.
- Requests from the client to a search service must be authenticated. To access search operations, the caller must have [role-based permissions](#) or provide an [API key](#) with the

request.

Related content

- [Reliability in Azure AI Search](#)
- [Design reliable Azure applications](#)

Security in Azure AI Search

09/25/2025

Azure AI Search provides comprehensive security controls across network access, data access, and data protection to meet enterprise requirements. As a solution architect, you should understand three key security domains:

- **Network traffic patterns and network security:** Inbound, outbound, and internal traffic.
- **Access control mechanisms:** API keys or Microsoft Entra ID with roles.
- **Data residency and protection:** Encryption in transit, in use with optional confidential computing, and at rest with optional double encryption.

A search service supports multiple network security topologies, from IP firewall restrictions for basic protection to private endpoints for complete network isolation. Optionally, use a network security perimeter to create a logical boundary around your Azure PaaS resources. For enterprise scenarios requiring granular permissions, you can implement document-level access controls. All security features integrate with Azure's compliance framework and support common enterprise patterns like multitenancy and cross-service authentication using managed identities.

This article details the implementation options for each security layer to help you design appropriate security architectures for development and production environments.

Network traffic patterns

An Azure AI Search service can be hosted in the Azure public cloud, an Azure private cloud, or a sovereign cloud (such as Azure Government). By default, for all cloud hosts, the search service is typically accessed by client applications over public network connections. While that pattern is predominant, it's not the only traffic pattern that you need to care about. Understanding all points of entry as well as outbound traffic is necessary background for securing your development and production environments.

Azure AI Search has three basic network traffic patterns:

- Inbound requests made by a user or client to the search service (the predominant pattern)
- Outbound requests issued by the search service to other services on Azure and elsewhere
- Internal service-to-service requests over the secure Microsoft backbone network

Inbound traffic

Inbound requests that target a search service endpoint include:

- Create, read, update, or delete indexes and other objects on the search service
- Load an index with search documents
- Query an index
- Run indexer or skillset jobs

The [REST APIs](#) describe the full range of inbound requests that are handled by a search service.

At a minimum, all inbound requests must be authenticated using either of these options:

- Key-based authentication (default). Inbound requests provide a valid API key.
- Role-based access control. Authorization is through Microsoft Entra identities and role assignments on your search service.

Additionally, you can add [network security features](#) to further restrict access to the endpoint. You can create either inbound rules in an IP firewall, or create private endpoints that fully shield your search service from the public internet.

Outbound traffic

Outbound requests can be secured and managed by you. Outbound requests originate from a search service to other applications. These requests are typically made by indexers for text-based and multimodal indexing, custom skills-based AI enrichment, and vectorizations at query time. Outbound requests include both read and write operations.

The following list is a full enumeration of the outbound requests for which you can configure secure connections. A search service makes requests on its own behalf, and on the behalf of an indexer or custom skill.

 [Expand table](#)

Operation	Scenario
Indexers	Connect to external data sources to retrieve data (read access). For more information, see Indexer access to content protected by Azure network security .
Indexers	Connect to Azure Storage for write operations to knowledge stores , cached enrichments , debug sessions .
Custom skills	Connect to Azure functions, Azure web apps, or other apps running external code that's hosted off-service. The request for external processing is sent during skillset execution.
Indexers and integrated	Connect to Azure OpenAI and a deployed embedding model, or it goes through a custom skill to connect to an embedding model that you provide. The search service

Operation	Scenario
vectorization	sends text to embedding models for vectorization during indexing.
Vectorizers	Connect to Azure OpenAI or other embedding models at query time to convert user text strings to vectors for vector search.
Knowledge agents	Connect to chat completion models for agentic retrieval query planning, and also for formulating answers grounded in search results. If you're implementing a basic RAG pattern , your query logic calls an external chat completion model for formulating an answer grounded in search results. For this pattern, the connection to the model uses the identity of your client or user. The search service identity isn't used for the connection. In contrast, if you use knowledge agents in a RAG retrieval pattern, the outbound request is made by the search service managed identity.
Search service	Connect to Azure Key Vault for customer-managed encryption keys used to encrypt and decrypt sensitive data.

Outbound connections can generally be made using a resource's full access connection string that includes a key or a database login, or a [managed identity](#) if you're using Microsoft Entra ID and role-based access.

To reach Azure resources behind a firewall, [create inbound rules on other Azure resources that admit search service requests](#).

To reach Azure resources protected by Azure Private Link, [create a shared private link](#) that an indexer uses to make its connection.

Exception for same-region search and storage services

If Azure Storage and Azure AI Search are in the same region, network traffic is routed through a private IP address and occurs over the Microsoft backbone network. Because private IP addresses are used, you can't configure IP firewalls or a private endpoint for network security.

Configure same-region connections using either of the following approaches:

- [Trusted service exception](#)
- [Resource instance rules](#)

Internal traffic

Internal requests are secured and managed by Microsoft. You can't configure or control these connections. If you're locking down network access, no action on your part is required because internal traffic isn't customer-configurable.

Internal traffic consists of:

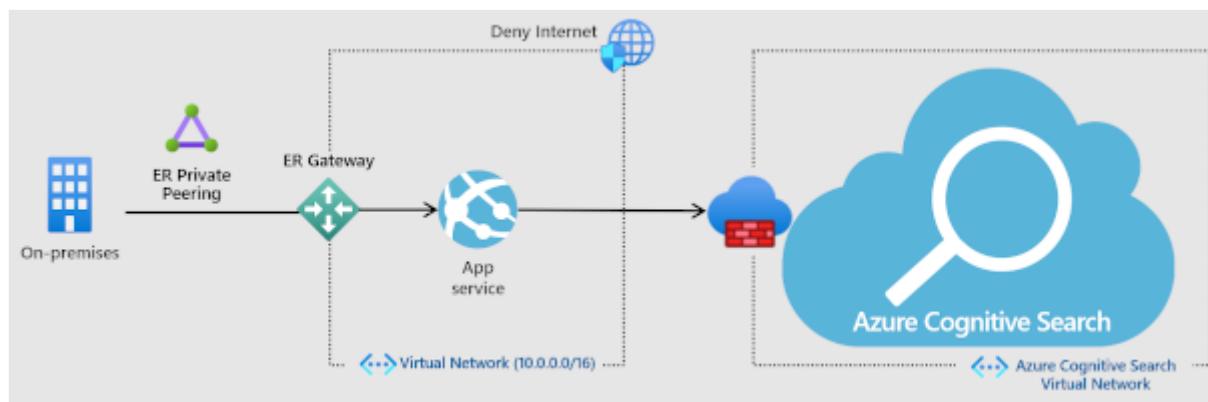
- Service-to-service calls for tasks like authentication and authorization through Microsoft Entra ID, resource logging sent to Azure Monitor, and [private endpoint connections](#) that utilize Azure Private Link.
- Requests for [built-in skills processing](#), with same-region requests directed to an internally hosted Azure AI multi-service resource used exclusively for built-in skills processing by Azure AI Search.
- Requests made to the various models that support [semantic ranking](#).

Network security

[Network security](#) protects resources from unauthorized access or attack by applying controls to network traffic. Azure AI Search supports networking features that can be your frontline of defense against unauthorized access.

Inbound connection through IP firewalls

A search service is provisioned with a public endpoint that allows access using a public IP address. To restrict which traffic comes through the public endpoint, create an inbound firewall rule that admits requests from a specific IP address or a range of IP addresses. All client connections must be made through an allowed IP address, or the connection is denied.



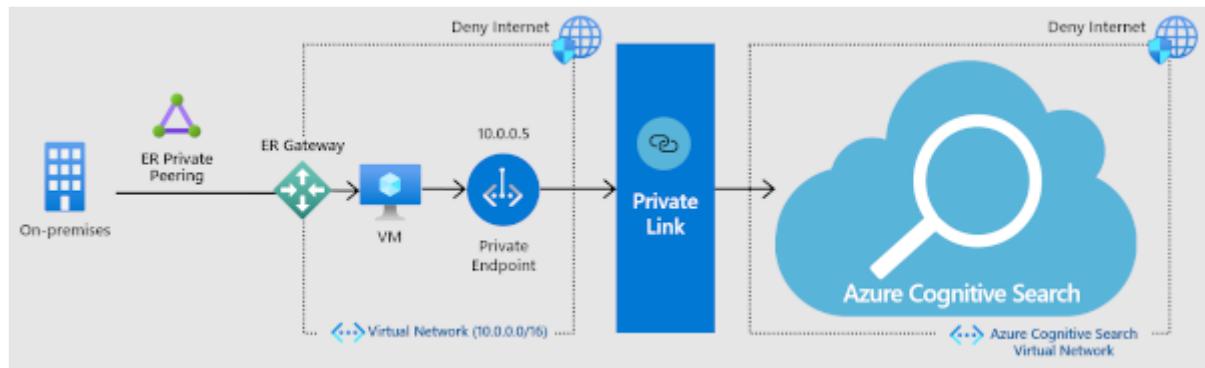
You can use the Azure portal to [configure firewall access](#).

Alternatively, you can use the management REST APIs. Starting with API version 2020-03-13, with the [IpRule](#) parameter, you can restrict access to your service by identifying IP addresses, individually or in a range, that you want to grant access to your search service.

Inbound connection to a private endpoint (network isolation, no Internet traffic)

For more stringent security, you can establish a [private endpoint](#) for Azure AI Search allows a client on a [virtual network](#) to securely access data in a search index over a [Private Link](#).

The private endpoint uses an IP address from the virtual network address space for connections to your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. A virtual network allows for secure communication among resources, with your on-premises network as well as the Internet.



While this solution is the most secure, using more services is an added cost so be sure you have a clear understanding of the benefits before diving in. For more information about costs, see the [pricing page](#). For more information about how these components work together, [watch this video](#). Coverage of the private endpoint option starts at 5:48 into the video. For instructions on how to set up the endpoint, see [Create a Private Endpoint for Azure AI Search](#).

Network security perimeter

A network security perimeter is a logical network boundary around your platform-as-a-service (PaaS) resources that are deployed outside of a virtual network. It establishes a perimeter for controlling public network access to resources like Azure AI Search, Azure Storage, and Azure OpenAI. You can grant exceptions through explicit access rules for inbound and outbound traffic. This approach helps prevent data exfiltration while maintaining necessary connectivity for your applications.

Inbound client connections and service-to-service connections occur within the boundary, which simplifies and strengthens your defenses against unauthorized access. It's common in Azure AI Search solutions to use multiple Azure resources. The following resources can all be joined to an [existing network security perimeter](#):

- [Azure AI Search](#)
- [Azure OpenAI](#)
- [Azure Storage](#)
- [Azure Monitor](#)

For a complete list of eligible services, see [Onboarded private link resources](#).

Authentication

Once a request is admitted to the search service, it must still undergo authentication and authorization that determines whether the request is permitted. Azure AI Search supports two approaches:

- [Microsoft Entra authentication](#) establishes the caller (and not the request) as the authenticated identity. An Azure role assignment determines authorization.
- [Key-based authentication](#) is performed on the request (not the calling app or user) through an API key, where the key is a string composed of randomly generated numbers and letters that prove the request is from a trustworthy source. Keys are required on every request. Submission of a valid key is considered proof the request originates from a trusted entity.

Reliance on API key-based authentication means that you should have a plan for regenerating the admin key at regular intervals, per Azure security best practices. There are a maximum of two admin keys per search service. For more information about securing and managing API keys, see [Create and manage api-keys](#).

Key-based authentication is the default for data plane operations (creating and using objects on the search service). You can use both authentication methods, or [disable an approach](#) that you don't want available on your search service.

Authorization

Azure AI Search provides authorization models for service management and content management.

Privileged access

On a new search service, existing role assignments at the subscription level are inherited by the search service, and only Owners and User Access Administrators can grant access.

Control plane operations (service or resource creation and management) tasks are exclusively authorized through [role assignments](#), with no ability to use key-based authentication for service administration.

Control plane operations include create, configure, or delete the service, and manage security. As such, Azure role assignments will determine who can perform those tasks, regardless of

whether they're using the [portal](#), [PowerShell](#), or the [Management REST APIs](#).

Three basic roles (Owner, Contributor, Reader) apply to search service administration.

 Note

Using Azure-wide mechanisms, you can lock a subscription or resource to prevent accidental or unauthorized deletion of your search service by users with admin rights. For more information, see [Lock resources to prevent unexpected deletion](#).

Authorize access to content

Data plane operations refers to the objects created and used on a search service.

- For role-based authorization, [use Azure role assignments](#) to establish read-write access to operations.
- For key-based authorization, [an API key](#) and a qualified endpoint determine access. An endpoint might be the service itself, the indexes collection, a specific index, a documents collection, or a specific document. When chained together, the endpoint, the operation (for example, a create request) and the type of key (admin or query) authorize access to content and operations.

Restricting access to indexes

Using Azure roles, you can [set permissions on individual indexes](#) as long as it's done programmatically.

Using keys, anyone with an [admin key](#) to your service can read, modify, or delete any index in the same service. For protection against accidental or malicious deletion of indexes, your in-house source control for code assets is the solution for reversing an unwanted index deletion or modification. Azure AI Search has failover within the cluster to ensure availability, but it doesn't store or execute your proprietary code used to create or load indexes.

For multitenancy solutions requiring security boundaries at the index level, it's common to handle index isolation in the middle tier in your application code. For more information about the multitenant use case, see [Design patterns for multitenant SaaS applications and Azure AI Search](#).

Restricting access to documents

User permissions at the document level, also known as *row-level security*, is available as a preview feature and depends on the data source. If content originates from [Azure Data Lake Storage \(ADLS\) Gen2](#) or [Azure blobs](#), user permission metadata that originates in Azure Storage is preserved in indexer-generated indexes and enforced at query time so that only authorized content is included in search results.

For other data sources, you can [push a document payload that includes user or group permission metadata](#), and those permissions are retained in indexed content and also enforced at query time. This capability is also in preview.

If you can't use preview features and you require permissioned access over content in search results, there's a technique for applying filters that include or exclude documents based on user identity. This workaround adds a string field in the data source that represents a group or user identity, which you can make filterable in your index. For more information about this pattern, see [Security trimming based on identity filters](#). For more information about document access, see [Document-level access control](#).

Data residency

When you set up a search service, you choose a region that determines where customer data is stored and processed. Each region exists within a [geography \(Geo\)](#) ↗ that often includes multiple regions (for example, Switzerland is a Geo that contains Switzerland North and Switzerland West). Azure AI Search might replicate your data to another region within the same Geo for durability and high availability. The service won't store or process customer data outside of your specified Geo unless you configure a feature that has a dependency on another Azure resource, and that resource is provisioned in a different region.

Currently, the only external resource that a search service writes to is Azure Storage. The storage account is one that you provide, and it could be in any region. A search service writes to Azure Storage if you use any of the following features:

- [enrichment cache](#)
- [debug session](#)
- [knowledge store](#)

For more information about data residency, see [data residency in Azure](#) ↗ .

Exceptions to data residency commitments

Object names appear in the telemetry logs used by Microsoft to provide support for the service. Object names are stored and processed outside of your selected region or location. Object names include the names of indexes and index fields, aliases, indexers, data sources,

skillsets, synonym maps, resources, containers, and key vault store. Customers shouldn't place any sensitive data in name fields or create applications designed to store sensitive data in these fields.

Telemetry logs are retained for one and a half years. During that period, Microsoft might access and reference object names under the following conditions:

- Diagnose an issue, improve a feature, or fix a bug. In this scenario, data access is internal only, with no third-party access.
- During support, this information might be used to provide quick resolution to issues and escalate product team if needed

Data protection

At the storage layer, data encryption is built in for all service-managed content saved to disk, including indexes, synonym maps, and the definitions of indexers, data sources, and skillsets. Service-managed encryption applies to both long-term data storage and temporary data storage.

Optionally, you can add customer-managed keys (CMK) for supplemental encryption of indexed content for double encryption of data at rest. For services created after August 1 2020, CMK encryption extends to short-term data on temporary disks.

Data in transit

For search service connections over the public internet, Azure AI Search listens on HTTPS port 443.

Azure AI Search supports TLS 1.2 and 1.3 for client-to-service channel encryption:

- TLS 1.3 is the default on newer client operating systems and versions of .NET.
- TLS 1.2 is the default on older systems, but you can [explicitly set TLS 1.3 on a client request](#).

Earlier versions of TLS (1.0 or 1.1) aren't supported.

For more information, see [TLS support in .NET Framework](#).

Data in use

By default, Azure AI Search deploys your search service on standard Azure infrastructure. This infrastructure encrypts data at rest and in transit, but it doesn't protect data while it's being

actively processed in memory.

Optionally, you can use the [Azure portal](#) or [Services - Create or Update \(REST API\)](#) to configure confidential computing during service creation. Confidential computing protects data in use from unauthorized access, including from Microsoft, through hardware attestation and encryption. For more information, see [Confidential computing use cases](#).

The following table compares both compute types.

[] [Expand table](#)

Compute type	Description	Limitations	Cost	Availability
Default	Processes data on standard VMs with built-in encryption for data at rest and in transit. No hardware-based isolation for data in use.	No limitations.	No change to the base cost of free or billable tiers.	Available in all regions.
Confidential	Processes data on confidential VMs (DCasv5 or DCesv5) in a hardware-based trusted execution environment. Isolates computations and memory from the host operating system and other VMs.	Disables or restricts agentic retrieval , semantic ranker , query rewrite , skillset execution , and indexers that run in the multitenant environment ¹ .	Adds a 10% surcharge to the base cost of billable tiers. For more information, see the pricing page ↗.	Available in some regions. For more information, see the list of supported regions .

¹ When you enable this compute type, indexers can only run in the private execution environment, meaning they run from the search clusters hosted on confidential computing.

(i) Important

We only recommend confidential computing for organizations whose compliance or regulatory requirements necessitate data-in-use protection. For daily usage, the default compute type suffices.

Data at rest

For data handled internally by the search service, the following table describes the [data encryption models](#). Some features, such as knowledge store, incremental enrichment, and

indexer-based indexing, read from or write to data structures in other Azure Services. Services that have a dependency on Azure Storage can use the [encryption features](#) of that technology.

[+] [Expand table](#)

Model	Keys	Requirements	Restrictions	Applies to
server-side encryption	Microsoft-managed keys	None (built-in)	None, available on all tiers, in all regions, for content created after January 24, 2018.	Content (indexes and synonym maps) and definitions (indexers, data sources, skillsets) on data disks and temporary disks
server-side encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in specific regions, for content created after August 1, 2020.	Content (indexes and synonym maps) and definitions (indexers, data sources, skillsets) on data disks
server-side full encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in all regions, on search services after May 13, 2021.	Content (indexes and synonym maps) and definitions (indexers, data sources, skillsets) on data disks and temporary disks

When you introduce CMK encryption, you're encrypting content twice. For the objects and fields noted in the previous section, content is first encrypted with your CMK, and secondly with the Microsoft-managed key. Content is doubly encrypted on data disks for long-term storage, and on temporary disks used for short-term storage.

Service-managed keys

Service-managed encryption is a Microsoft-internal operation that uses 256-bit [AES encryption](#). It occurs automatically on all indexing, including on incremental updates to indexes that aren't fully encrypted (created before January 2018).

Service-managed encryption applies to all content on long-term and short-term storage.

Customer-managed keys (CMK)

Customers use CMK for two reasons: extra protection, and the ability to revoke keys, preventing access to content.

Customer-managed keys require another billable service, Azure Key Vault, which can be in a different region, but under the same Azure tenant, as Azure AI Search.

CMK support was rolled out in two phases. If you created your search service during the first phase, CMK encryption was restricted to long-term storage and specific regions. Services created in the second phase can use CMK encryption in any region. As part of the second wave rollout, content is CMK-encrypted on both long-term and short-term storage.

- The first rollout was on August 1, 2020 and included the following five regions. Search services created in the following regions supported CMK for data disks, but not temporary disks:
 - West US 2
 - East US
 - South Central US
 - US Gov Virginia
 - US Gov Arizona
- The second rollout on May 13, 2021 added encryption for temporary disks and extended CMK encryption to [all supported regions](#).

If you're using CMK from a service created during the first rollout and you also want CMK encryption over temporary disks, you need to create a new search service in your region of choice and redeploy your content. To determine your service creation date, see [Check your service creation or upgrade date](#).

Enabling CMK encryption will increase index size and degrade query performance. Based on observations to date, you can expect to see an increase of 30-60 percent in query times, although actual performance will vary depending on the index definition and types of queries. Because of the negative performance impact, we recommend that you only enable this feature on indexes that really require it. For more information, see [Configure customer-managed encryption keys in Azure AI Search](#).

Logging and monitoring

Azure AI Search doesn't log user identities so you can't refer to logs for information about a specific user. However, the service does log create-read-update-delete operations, which you might be able to correlate with other logs to understand the agency of specific actions.

Using alerts and the logging infrastructure in Azure, you can pick up on query volume spikes or other actions that deviate from expected workloads. For more information about setting up logs, see [Collect and analyze log data](#) and [Monitor query requests](#).

Compliance and governance

Azure AI Search participates in regular audits, and has been certified against many global, regional, and industry-specific standards for both the public cloud and Azure Government. For the complete list, download the [Microsoft Azure Compliance Offerings whitepaper](#) from the official Audit reports page.

We recommend that you regularly review [Azure AI Search compliance certifications and documentation](#) to ensure alignment with your regulatory requirements.

Use Azure Policy

For compliance, you can use [Azure Policy](#) to implement the high-security best practices of [Microsoft cloud security benchmark](#). The Microsoft cloud security benchmark is a collection of security recommendations, codified into security controls that map to key actions you should take to mitigate threats to services and data. There are currently 12 security controls, including [Network Security](#), Logging and Monitoring, and [Data Protection](#).

Azure Policy is a capability built into Azure that helps you manage compliance for multiple standards, including those of Microsoft cloud security benchmark. For well-known benchmarks, Azure Policy provides built-in definitions that provide both criteria and an actionable response that addresses noncompliance.

For Azure AI Search, there's currently one built-in definition. It's for resource logging. You can assign a policy that identifies search services that are missing resource logging, and then turn it on. For more information, see [Azure Policy Regulatory Compliance controls for Azure AI Search](#).

Use tags

Apply metadata tags to categorize search services based on data sensitivity and compliance requirements. This facilitates proper governance and security controls. For more information, see [Use tags to organize your Azure resources](#) and [General guidance – Organize Azure resources using tags](#).

Learn more

- [Azure security fundamentals](#)
- [Azure Security](#)
- [Microsoft Defender for Cloud](#)

We also recommend the following [video on security features](#). It's several years old and doesn't cover newer features, but it covers these features: CMK, IP firewalls, and private link. If you use those features, you might find this video helpful.

Quickstart: Connect to a search service

In this quickstart, you use role-based access control (RBAC) and Microsoft Entra ID to establish a keyless connection to your Azure AI Search service. You then use Python in Visual Studio Code to interact with your service.

Keyless connections provide enhanced security through granular permissions and identity-based authentication. We don't recommend hard-coded API keys, but if you prefer them, see [Connect to Azure AI Search using keys](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#) in any region or tier.
- The [Azure CLI](#) for keyless authentication with Microsoft Entra ID.
- [Visual Studio Code](#) with the [Python extension](#) and [Jupyter package](#).

Configure role-based access

In this section, you enable RBAC on your Azure AI Search service and assign the necessary roles for creating, loading, and querying search objects. For more information about these steps, see [Connect to Azure AI Search using roles](#).

To configure access:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Settings > Keys**.
3. Select **Role-based access control** or **Both** if you need time to transition clients to RBAC.

my-search-service | Keys

Search service

API Access control

API keys

Role-based access control (selected)

Both

Manage admin keys

Primary admin key

Secondary admin key

Regenerate

Regenerate

Manage query keys

Add Delete

Keys

Scale

Search traffic analytics

4. From the left pane, select **Access control (IAM)**.

5. Select **Add > Add role assignment**.

my-search-service | Access control (IAM)

Search service

+ Add

Add role assignment

Check access

Role assignments

Roles

Deny assignments

Classic administrators

My access

View my access

Check access

Check access

6. Assign the **Search Service Contributor** role to your user account or managed identity.

7. Repeat the role assignment for **Search Index Data Contributor**.

Get service information

In this section, you retrieve the subscription ID and endpoint of your Azure AI Search service. If you only have one subscription, skip the subscription ID and only retrieve the endpoint. You use these values in the remaining sections of this quickstart.

To get your service information:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview**.
3. Make a note of the subscription ID and endpoint.

The screenshot shows the Azure portal interface for a search service named 'my-search-service'. The left sidebar has options like Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Search management, Indexes, Indexers, Data sources, and Aliases. The 'Overview' tab is selected. In the main content area, under the 'Essentials' section, there are several details: Resource group (move) : my-resource-group, Location (move) : East US, Subscription (move) : my-subscription, Subscription ID : aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e (highlighted with a red box), Status : Running, Date created : May 1, 2025, 12:18:59 PM, and Tags (edit) : Add tags. To the right, there are more details: Url : https://my-search-service.search.windows.net, Pricing tier : Basic, Replicas : 1 (No SLA), Partitions : 1, and Search units : 1. At the bottom, there are links for Get started, Properties, Usage, and Monitoring, and a magnifying glass icon.

Sign in to Azure

Before you connect to your Azure AI Search service, use the Azure CLI to sign in to the subscription that contains your service. This step establishes your Microsoft Entra identity, which `DefaultAzureCredential` uses to authenticate requests in the next section.

To sign in:

1. On your local system, open a command-line tool.
2. Check the active subscription and tenant in your local environment.

The screenshot shows a terminal window with the title 'Azure CLI'. Inside the terminal, the command `az account show` is typed and highlighted in blue, indicating it is being run or has been run.

3. If the active subscription and tenant aren't valid for your search service, run the following commands to update their values. You can find the subscription ID on the search service **Overview** page in the Azure portal. To find the tenant ID, select the name of your subscription on the **Overview** page, and then locate the **Parent management group** value.

Azure CLI

```
az account set --subscription <your-subscription-id>  
az login --tenant <your-tenant-id>
```

Connect to Azure AI Search

! Note

This section illustrates the basic Python pattern for keyless connections. For comprehensive guidance, see a specific quickstart or tutorial, such as [Quickstart: Use agentic retrieval in Azure AI Search](#).

You can use Python notebooks in Visual Studio Code to send requests to your Azure AI Search service. For request authentication, use the `DefaultAzureCredential` class from the Azure Identity library.

To connect using Python:

1. On your local system, open Visual Studio Code.
2. Create a `.ipynb` file.
3. Create a code cell to install the `azure-identity` and `azure-search-documents` libraries.

Python

```
pip install azure-identity azure-search-documents
```

4. Create another code cell to authenticate and connect to your search service.

Python

```
from azure.identity import DefaultAzureCredential  
from azure.search.documents.indexes import SearchIndexClient  
  
service_endpoint = "PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE"  
credential = DefaultAzureCredential()  
client = SearchIndexClient(endpoint = service_endpoint, credential =  
    credential)  
  
# List existing indexes  
indexes = client.list_indexes()  
  
for index in indexes:
```

```
index_dict = index.as_dict()
print(json.dumps(index_dict, indent = 2))
```

5. Set `service_endpoint` to the value you obtained in [Get service information](#).

6. Select **Run All** to run both code cells.

The output should list the existing indexes (if any) on your search service, indicating a successful connection.

Troubleshoot 401 errors

If you encounter a 401 error, follow these troubleshooting steps:

- Revisit [Configure role-based access](#). Your search service must have **Role-based access control** or **Both** enabled. Policies at the subscription or resource group level might override your role assignments.
- Revisit [Sign in to Azure](#). You must sign in to the subscription that contains your search service.
- Make sure your endpoint variable has surrounding quotes.
- If all else fails, restart your device to remove cached tokens and then repeat the steps in this quickstart, starting with [Sign in to Azure](#).

Related content

- [Configure a managed identity in Azure AI Search](#)
- [Connect your app to Azure AI Search using identities](#)
- [Configure network access and firewall rules for Azure AI Search](#)

Last updated on 11/20/2025

Quickstart: Use agentic retrieval in Azure AI Search

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this quickstart, you use [agentic retrieval](#) to create a conversational search experience powered by documents indexed in Azure AI Search and a large language model (LLM) from Azure OpenAI in Foundry Models.

A *knowledge base* orchestrates agentic retrieval by decomposing complex queries into subqueries, running the subqueries against one or more *knowledge sources*, and returning results with metadata. By default, the knowledge base outputs raw content from your sources, but this quickstart uses the answer synthesis output mode for natural-language answer generation.

Although you can provide your own data, this quickstart uses [sample JSON documents](#) from NASA's Earth at Night e-book. The documents describe general science topics and images of Earth at night as observed from space.

! Tip

Want to get started right away? See the [azure-search-dotnet-samples](#) GitHub repository.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#) in any region that provides agentic retrieval.
- A [Microsoft Foundry project](#) and resource. When you create a project, the resource is automatically created.
- The [Azure CLI](#) for keyless authentication with Microsoft Entra ID.
- [Visual Studio Code](#).

Configure access

Before you begin, make sure you have permissions to access content and operations. We recommend Microsoft Entra ID for authentication and role-based access for authorization. You must be an **Owner** or **User Access Administrator** to assign roles. If roles aren't feasible, use [key-based authentication](#) instead.

To configure access for this quickstart, select both of the following tabs.

Azure AI Search

Azure AI Search provides the agentic retrieval pipeline. Configure access for yourself and your search service to read and write data, interact with Foundry, and run the pipeline.

On your Azure AI Search service:

1. [Enable role-based access.](#)
2. [Create a system-assigned managed identity.](#)
3. [Assign the following roles](#) to yourself.
 - **Search Service Contributor**
 - **Search Index Data Contributor**
 - **Search Index Data Reader**

ⓘ Important

Agentic retrieval has two token-based billing models:

- Billing from Azure AI Search for agentic retrieval.
- Billing from Azure OpenAI for query planning and answer synthesis.

For more information, see [Availability and pricing of agentic retrieval](#).

Get endpoints

Each Azure AI Search service and Foundry resource has an *endpoint*, which is a unique URL that identifies and provides network access to the resource. In a later section, you specify these endpoints to connect to your resources programmatically.

To get the endpoints for this quickstart, select both of the following tabs.

Azure AI Search

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview**.
3. Make a note of the endpoint, which should look like `https://my-service.search.windows.net`.

Deploy models

To use agentic retrieval, you must deploy two Azure OpenAI models to your Foundry project:

- An embedding model for text-to-vector conversion. This quickstart uses `text-embedding-3-large`, but you can use any `text-embedding` model.
- An LLM for query planning and answer generation. This quickstart uses `gpt-5-mini`, but you can use any [supported LLM for agentic retrieval](#).

For deployment instructions, see [Deploy Azure OpenAI models with Foundry](#).

Set up the environment

To set up the console application for this quickstart:

1. Create a folder named `quickstart-agentic-retrieval` to contain the application.
2. Open the folder in Visual Studio Code.
3. Select **Terminal > New Terminal**, and then run the following command to create a console application.

Console

```
dotnet new console
```

4. Install the [Azure AI Search client library](#) for .NET.

Console

```
dotnet add package Azure.Search.Documents --version 11.8.0-beta.1
```

5. Install the `dotenv.net` package to load environment variables from a `.env` file.

Console

```
dotnet add package dotenv.net
```

6. For keyless authentication with Microsoft Entra ID, install the [Azure.Identity](#) package.

Console

```
dotnet add package Azure.Identity
```

7. For keyless authentication with Microsoft Entra ID, sign in to your Azure account. If you have multiple subscriptions, select the one that contains your Azure AI Search service and Foundry project.

Console

```
az login
```

Run the code

To create and run the agentic retrieval pipeline:

1. Create a file named `.env` in the `quickstart-agentic-retrieval` folder.
2. Paste the following environment variables into the `.env` file.

```
SEARCH_ENDPOINT = PUT-YOUR-SEARCH-SERVICE-URL-HERE
AOAI_ENDPOINT = PUT-YOUR-AOAI-FOUNDRY-URL-HERE
```

3. Set `SEARCH_ENDPOINT` and `AOAI_ENDPOINT` to the values you obtained in [Get endpoints](#).
4. Paste the following code into the `Program.cs` file.

C#

```
using dotenv.net;
using System.Text.Json;
using Azure.Identity;
using Azure.Search.Documents;
```

```
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.KnowledgeBases;
using Azure.Search.Documents.KnowledgeBases.Models;

namespace AzureSearch.Quickstart
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // Load environment variables from the .env file
            // Ensure your .env file is in the same directory with the required
variables
            DotEnv.Load();

            string searchEndpoint =
Environment.GetEnvironmentVariable("SEARCH_ENDPOINT")
                ?? throw new InvalidOperationException("SEARCH_ENDPOINT isn't
set.");
            string aoaiEndpoint =
Environment.GetEnvironmentVariable("AOAI_ENDPOINT")
                ?? throw new InvalidOperationException("AOAI_ENDPOINT isn't
set.");

            string aoaiEmbeddingModel = "text-embedding-3-large";
            string aoaiEmbeddingDeployment = "text-embedding-3-large";
            string aoaiGptModel = "gpt-5-mini";
            string aoaiGptDeployment = "gpt-5-mini";

            string indexName = "earth-at-night";
            string knowledgeSourceName = "earth-knowledge-source";
            string knowledgeBaseName = "earth-knowledge-base";

            var credential = new DefaultAzureCredential();

            // Define fields for the index
            var fields = new List<SearchField>
            {
                new SimpleField("id", SearchFieldDataType.String) { IsKey =
true, IsFilterable = true, IsSortable = true, IsFacetable = true },
                new SearchField("page_chunk", SearchFieldDataType.String) {
IsFilterable = false, IsSortable = false, IsFacetable = false },
                new SearchField("page_embedding_text_3_large",
SearchFieldDataType.Collection(SearchFieldDataType.Single)) {
VectorSearchDimensions = 3072, VectorSearchProfileName = "hnsw_text_3_large" },
                new SimpleField("page_number", SearchFieldDataType.Int32) {
IsFilterable = true, IsSortable = true, IsFacetable = true }
            };

            // Define a vectorizer
            var vectorizer = new AzureOpenAIVectorizer(vectorizerName:
"azure_openai_text_3_large")
            {
                Parameters = new AzureOpenAIVectorizerParameters
```

```
{
    ResourceUri = new Uri(aoaiEndpoint),
    DeploymentName = aoaiEmbeddingDeployment,
    ModelName = aoaiEmbeddingModel
}
};

// Define a vector search profile and algorithm
var vectorSearch = new VectorSearch()
{
    Profiles =
    {
        new VectorSearchProfile(
            name: "hnsw_text_3_large",
            algorithmConfigurationName: "alg"
        )
        {
            VectorizerName = "azure_openai_text_3_large"
        }
    },
    Algorithms =
    {
        new HnswAlgorithmConfiguration(name: "alg")
    },
    Vectorizers =
    {
        vectorizer
    }
};

// Define a semantic configuration
var semanticConfig = new SemanticConfiguration(
    name: "semantic_config",
    prioritizedFields: new SemanticPrioritizedFields
    {
        ContentFields = { new SemanticField("page_chunk") }
    }
);

var semanticSearch = new SemanticSearch()
{
    DefaultConfigurationName = "semantic_config",
    Configurations = { semanticConfig }
};

// Create the index
var index = new SearchIndex(indexName)
{
    Fields = fields,
    VectorSearch = vectorSearch,
    SemanticSearch = semanticSearch
};

// Create the index client, deleting and recreating the index if it
exists
```

```
        var indexClient = new SearchIndexClient(new Uri(searchEndpoint),
credential);
        await indexClient.CreateOrUpdateIndexAsync(index);
        Console.WriteLine($"Index '{indexName}' created or updated
successfully.");

        // Upload sample documents from the GitHub URL
        string url = "https://raw.githubusercontent.com/Azure-
Samples/azure-search-sample-data/refs/heads/main/nasa-e-book/earth-at-night-
json/documents.json";
        var httpClient = new HttpClient();
        var response = await httpClient.GetAsync(url);
        response.EnsureSuccessStatusCode();
        var json = await response.Content.ReadAsStringAsync();
        var documents = JsonSerializer.Deserialize<List<Dictionary<string,
object>>>(json);
        var searchClient = new SearchClient(new Uri(searchEndpoint),
indexName, credential);
        var searchIndexingBufferedSender = new
SearchIndexingBufferedSender<Dictionary<string, object>>(
            searchClient,
            new SearchIndexingBufferedSenderOptions<Dictionary<string,
object>>
{
            KeyFieldAccessor = doc => doc["id"].ToString(),
        }
    );

        await searchIndexingBufferedSender.UploadDocumentsAsync(documents);
        await searchIndexingBufferedSender.FlushAsync();
        Console.WriteLine($"Documents uploaded to index '{indexName}'
successfully.");

        // Create a knowledge source
        var indexKnowledgeSource = new SearchIndexKnowledgeSource(
            name: knowledgeSourceName,
            searchIndexParameters: new
SearchIndexKnowledgeSourceParameters(searchIndexName: indexName)
        {
            SourceDataFields = { new SearchIndexFieldReference(name:
"id"), new SearchIndexFieldReference(name: "page_chunk"), new
SearchIndexFieldReference(name: "page_number") }
        }
    );

        await
indexClient.CreateOrUpdateKnowledgeSourceAsync(indexKnowledgeSource);
        Console.WriteLine($"Knowledge source '{knowledgeSourceName}' 
created or updated successfully.");

        // Create a knowledge base
        var openAiParameters = new AzureOpenAIVectorizerParameters
{
            ResourceUri = new Uri(aoaiEndpoint),
            DeploymentName = aoaiGptDeployment,
```

```

        ModelName = aoaiGptModel
    };

    var model = new
KnowledgeBaseAzureOpenAIModel(azureOpenAIParameters: openAiParameters);

    var knowledgeBase = new KnowledgeBase(
        name: knowledgeBaseName,
        knowledgeSources: new KnowledgeSourceReference[] { new
KnowledgeSourceReference(knowledgeSourceName) }
    )
{
    RetrievalReasoningEffort = new
KnowledgeRetrievalLowReasoningEffort(),
    AnswerInstructions = "Provide a two sentence concise and
informative answer based on the retrieved documents.",
    Models = { model }
};

    await indexClient.CreateOrUpdateKnowledgeBaseAsync(knowledgeBase);
    Console.WriteLine($"Knowledge base '{knowledgeBaseName}' created or
updated successfully.");

    // Set up messages
    string instructions = @"A Q&A agent that can answer questions about
the Earth at night.
    If you don't have the answer, respond with ""I don't know"".;

    var messages = new List<Dictionary<string, string>>
{
    new Dictionary<string, string>
    {
        { "role", "system" },
        { "content", instructions }
    }
};

    // Run agentic retrieval
    var baseClient = new KnowledgeBaseRetrievalClient(
        endpoint: new Uri(searchEndpoint),
        knowledgeBaseName: knowledgeBaseName,
        tokenCredential: new DefaultAzureCredential()
    );

    string query = @"Why do suburban belts display larger December
brightening than urban cores even though absolute light levels are higher
downtown? Why is the Phoenix nighttime street grid is so sharply visible from
space, whereas large stretches of the interstate between midwestern cities
remain comparatively dim?";

    messages.Add(new Dictionary<string, string>
{
    { "role", "user" },
    { "content", query }
});

```

```
Console.WriteLine($"Running the query...{query}");
var retrievalRequest = new KnowledgeBaseRetrievalRequest();
foreach (Dictionary<string, string> message in messages) {
    if (message["role"] != "system") {
        retrievalRequest.Messages.Add(new
KnowledgeBaseMessage(content: new[] { new
KnowledgeBaseMessageTextContent(message[ "content" ]) }) { Role = message[ "role" ]
});}
    }
    retrievalRequest.RetrievalReasoningEffort = new
KnowledgeRetrievalLowReasoningEffort();
    var retrievalResult = await
baseClient.RetrieveAsync(retrievalRequest);

    messages.Add(new Dictionary<string, string>
{
    { "role", "assistant" },
    { "content", (retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent)!.Text }
});

    // Print the response, activity, and references
    Console.WriteLine("Response:");
    Console.WriteLine((retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent)!.Text);

    Console.WriteLine("Activity:");
    foreach (var activity in retrievalResult.Value.Activity)
    {
        Console.WriteLine($"Activity Type: {activity.GetType().Name}");
        string activityJson = JsonSerializer.Serialize(
            activity,
            activity.GetType(),
            new JsonSerializerOptions { WriteIndented = true })
        );
        Console.WriteLine(activityJson);
    }

    Console.WriteLine("References:");
    foreach (var reference in retrievalResult.Value.References)
    {
        Console.WriteLine($"Reference Type:
{reference.GetType().Name}");
        string referenceJson = JsonSerializer.Serialize(
            reference,
            reference.GetType(),
            new JsonSerializerOptions { WriteIndented = true })
        );
        Console.WriteLine(referenceJson);
    }

    // Continue the conversation
    string nextQuery = "How do I find lava at night?";
```

```

        Console.WriteLine($"Continue the conversation with this query:
{nextQuery}");
        messages.Add(new Dictionary<string, string>
        {
            { "role", "user" },
            { "content", nextQuery }
        });

        retrievalRequest = new KnowledgeBaseRetrievalRequest();
        foreach (Dictionary<string, string> message in messages) {
            if (message["role"] != "system") {
                retrievalRequest.Messages.Add(new
KnowledgeBaseMessage(content: new[] { new
KnowledgeBaseMessageTextContent(message[ "content" ]) }) { Role = message[ "role" ]
});}
        }
        retrievalRequest.RetrievalReasoningEffort = new
KnowledgeRetrievalLowReasoningEffort();
        retrievalResult = await baseClient.RetrieveAsync(retrievalRequest);

        messages.Add(new Dictionary<string, string>
{
    { "role", "assistant" },
    { "content", (retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent)!.Text }
});

// Print the new response, activity, and references
Console.WriteLine("Response:");
Console.WriteLine((retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent)!.Text);

Console.WriteLine("Activity:");
foreach (var activity in retrievalResult.Value.Activity)
{
    Console.WriteLine($"Activity Type: {activity.GetType().Name}");
    string activityJson = JsonSerializer.Serialize(
        activity,
        activity.GetType(),
        new JsonSerializerOptions { WriteIndented = true })
    );
    Console.WriteLine(activityJson);
}

Console.WriteLine("References:");
foreach (var reference in retrievalResult.Value.References)
{
    Console.WriteLine($"Reference Type:
{reference.GetType().Name}");
    string referenceJson = JsonSerializer.Serialize(
        reference,
        reference.GetType(),
        new JsonSerializerOptions { WriteIndented = true })
    );
}

```

```

        Console.WriteLine(referenceJson);
    }

    // Clean up resources
    await indexClient.DeleteKnowledgeBaseAsync(knowledgeBaseName);
    Console.WriteLine($"Knowledge base '{knowledgeBaseName}' deleted
successfully.");

    await indexClient.DeleteKnowledgeSourceAsync(knowledgeSourceName);
    Console.WriteLine($"Knowledge source '{knowledgeSourceName}' 
deleted successfully.");

    await indexClient.DeleteIndexAsync(indexName);
    Console.WriteLine($"Index '{indexName}' deleted successfully.");
}
}
}

```

5. Build and run the application.

Console

dotnet run

Output

The output of the application should be similar to the following:

```

Index 'earth-at-night' created or updated successfully.
Documents uploaded to index 'earth-at-night' successfully.
Knowledge source 'earth-knowledge-source' created or updated successfully.
Knowledge base 'earth-knowledge-base' created or updated successfully.
Response:
Suburban belts show larger December brightening because holiday displays concentrate
in suburbs and outskirts where there is more yard space and many single-family homes
[ref_id:5], while urban cores—already having higher absolute light levels—tend to
show smaller relative increases (central areas typically brighten ~20–30%)
[ref_id:8][ref_id:5]. Phoenix's nighttime street grid is sharply visible because the
metropolitan area is laid out on a regular, continuously lit grid with bright
commercial and industrial nodes along major corridors like Grand Avenue [ref_id:0]
[ref_id:3], whereas long interstate stretches between Midwestern cities cross
sparsely populated or rural regions with far fewer continuous roadside lights and so
appear comparatively dim [ref_id:8].
Activity:
Activity Type: KnowledgeBaseModelQueryPlanningActivityRecord
{
    "InputTokens": 1350,
    "OutputTokens": 1314,
    "Id": 0,
}
```

```
"ElapsedMs": 14162,
"Error": null
}
Activity Type: KnowledgeBaseSearchIndexActivityRecord
{
  "SearchIndexArguments": {
    "Search": "Causes of December brightening in satellite nightlights: why suburban belts show larger relative December brightening than urban cores (roles of holiday residential lighting, snow albedo, urban heat island, commercial lighting patterns)",
    "Filter": null,
    "SourceDataFields": [],
    "SearchFields": [],
    "SemanticConfigurationName": null
  },
  "KnowledgeSourceName": "earth-knowledge-source",
  "QueryTime": "2025-11-05T21:56:26.747+00:00",
  "Count": 19,
  "Id": 1,
  "ElapsedMs": 537,
  "Error": null
}
Activity Type: KnowledgeBaseSearchIndexActivityRecord
{
  "SearchIndexArguments": {
    "Search": "Why is Phoenix\u00e9s nighttime street grid so sharply visible from space? (effects of streetlight density, luminaire type/aiming, spacing, urban grid layout, traffic vs roadway lighting)",
    "Filter": null,
    "SourceDataFields": [],
    "SearchFields": [],
    "SemanticConfigurationName": null
  },
  "KnowledgeSourceName": "earth-knowledge-source",
  "QueryTime": "2025-11-05T21:56:27.182+00:00",
  "Count": 7,
  "Id": 2,
  "ElapsedMs": 434,
  "Error": null
}
Activity Type: KnowledgeBaseSearchIndexActivityRecord
{
  "SearchIndexArguments": {
    "Search": "How do satellite nightlight sensor characteristics (VIIRS DNB, DMSP-OLS) \u2014 spatial resolution, dynamic range, saturation, blooming \u2014 affect observed brightness and structure of urban cores, suburbs, and long interstate stretches?",
    "Filter": null,
    "SourceDataFields": [],
    "SearchFields": [],
    "SemanticConfigurationName": null
  },
  "KnowledgeSourceName": "earth-knowledge-source",
  "QueryTime": "2025-11-05T21:56:27.786+00:00",
  "Count": 23,
```

```
"Id": 3,
"ElapsedMs": 604,
"Error": null
}
Activity Type: KnowledgeBaseAgenticReasoningActivityRecord
{
  "ReasoningTokens": 70232,
  "RetrievalReasoningEffort": {},
  "Id": 4,
  "ElapsedMs": null,
  "Error": null
}
Activity Type: KnowledgeBaseModelAnswerSynthesisActivityRecord
{
  "InputTokens": 7467,
  "OutputTokens": 1710,
  "Id": 5,
  "ElapsedMs": 26663,
  "Error": null
}
Results:
Reference Type: KnowledgeBaseSearchIndexReference
{
  "DocKey": "earth_at_night_508_page_104_verbalized",
  "Id": "0",
  "ActivitySource": 2,
  "SourceData": {},
  "RerankerScore": 2.6344998
}
Reference Type: KnowledgeBaseSearchIndexReference
{
  "DocKey": "earth_at_night_508_page_194_verbalized",
  "Id": "1",
  "ActivitySource": 3,
  "SourceData": {},
  "RerankerScore": 2.630955
}
Reference Type: KnowledgeBaseSearchIndexReference
{
  "DocKey": "earth_at_night_508_page_105_verbalized",
  "Id": "3",
  "ActivitySource": 2,
  "SourceData": {},
  "RerankerScore": 2.5884187
}
Reference Type: KnowledgeBaseSearchIndexReference
{
  "DocKey": "earth_at_night_508_page_189_verbalized",
  "Id": "4",
  "ActivitySource": 3,
  "SourceData": {},
  "RerankerScore": 2.465418
}
Reference Type: KnowledgeBaseSearchIndexReference
{
```

```

    "DocKey": "earth_at_night_508_page_193_verbalized",
    "Id": "6",
    "ActivitySource": 3,
    "SourceData": {},
    "RerankerScore": 2.4560246
}
Reference Type: KnowledgeBaseSearchIndexReference
{
    "DocKey": "earth_at_night_508_page_174_verbalized",
    "Id": "2",
    "ActivitySource": 1,
    "SourceData": {},
    "RerankerScore": 2.3254027
}
Reference Type: KnowledgeBaseSearchIndexReference
{
    "DocKey": "earth_at_night_508_page_176_verbalized",
    "Id": "5",
    "ActivitySource": 1,
    "SourceData": {},
    "RerankerScore": 2.257256
}
Reference Type: KnowledgeBaseSearchIndexReference
{
    "DocKey": "earth_at_night_508_page_177_verbalized",
    "Id": "7",
    "ActivitySource": 1,
    "SourceData": {},
    "RerankerScore": 2.1968744
}
Reference Type: KnowledgeBaseSearchIndexReference
{
    "DocKey": "earth_at_night_508_page_125_verbalized",
    "Id": "8",
    "ActivitySource": 2,
    "SourceData": {},
    "RerankerScore": 2.086579
}
Response:
... // Trimmed for brevity
Activity:
... // Trimmed for brevity
References:
... // Trimmed for brevity
Knowledge base 'earth-knowledge-base' deleted successfully.
Knowledge source 'earth-knowledge-source' deleted successfully.
Index 'earth-at-night' deleted successfully.

```

Understand the code

Now that you've run the code, let's break down the key steps:

1. [Create a search index](#)
2. [Upload documents to the index](#)
3. [Create a knowledge source](#)
4. [Create a knowledge base](#)
5. [Set up messages](#)
6. [Run the retrieval pipeline](#)
7. [Continue the conversation](#)

Create a search index

In Azure AI Search, an index is a structured collection of data. The following code defines an index named `earth-at-night`, which you previously specified using the `indexName` variable.

The index schema contains fields for document identification and page content, embeddings, and numbers. The schema also includes configurations for semantic ranking and vector search, which uses your `text-embedding-3-large` deployment to vectorize text and match documents based on semantic or conceptual similarity.

C#

```
// Define fields for the index
var fields = new List<SearchField>
{
    new SimpleField("id", SearchFieldDataType.String) { IsKey = true, IsFilterable = true, IsSortable = true, IsFacetable = true },
    new SearchField("page_chunk", SearchFieldDataType.String) { IsFilterable = false, IsSortable = false, IsFacetable = false },
    new SearchField("page_embedding_text_3_large",
        SearchFieldDataType.Collection(SearchFieldDataType.Single)) { VectorSearchDimensions = 3072, VectorSearchProfileName = "hnsw_text_3_large" },
    new SimpleField("page_number", SearchFieldDataType.Int32) { IsFilterable = true, IsSortable = true, IsFacetable = true }
};

// Define a vectorizer
var vectorizer = new AzureOpenAIVectorizer(vectorizerName:
"azure_openai_text_3_large")
{
    Parameters = new AzureOpenAIVectorizerParameters
    {
        ResourceUri = new Uri(aoaiEndpoint),
        DeploymentName = aoaiEmbeddingDeployment,
        ModelName = aoaiEmbeddingModel
    }
};

// Define a vector search profile and algorithm
var vectorSearch = new VectorSearch()
{
```

```

Profiles =
{
    new VectorSearchProfile(
        name: "hnsw_text_3_large",
        algorithmConfigurationName: "alg"
    )
    {
        VectorizerName = "azure_openai_text_3_large"
    }
},
Algorithms =
{
    new HnswAlgorithmConfiguration(name: "alg")
},
Vectorizers =
{
    vectorizer
}
};

// Define a semantic configuration
var semanticConfig = new SemanticConfiguration(
    name: "semantic_config",
    prioritizedFields: new SemanticPrioritizedFields
    {
        ContentFields = { new SemanticField("page_chunk") }
    }
);

var semanticSearch = new SemanticSearch()
{
    DefaultConfigurationName = "semantic_config",
    Configurations = { semanticConfig }
};

// Create the index
var index = new SearchIndex(indexName)
{
    Fields = fields,
    VectorSearch = vectorSearch,
    SemanticSearch = semanticSearch
};

// Create the index client, deleting and recreating the index if it exists
var indexClient = new SearchIndexClient(new Uri(searchEndpoint), credential);
await indexClient.CreateOrUpdateIndexAsync(index);
Console.WriteLine($"Index '{indexName}' created or updated successfully.");

```

Upload documents to the index

Currently, the `earth-at-night` index is empty. The following code populates the index with JSON documents from [NASA's Earth at Night e-book](#). As required by Azure AI Search, each

document conforms to the fields and data types defined in the index schema.

C#

```
// Upload sample documents from the GitHub URL
string url = "https://raw.githubusercontent.com/Azure-Samples/azure-search-sample-data/refs/heads/main/nasa-e-book/earth-at-night-json/documents.json";
var httpClient = new HttpClient();
var response = await httpClient.GetAsync(url);
response.EnsureSuccessStatusCode();
var json = await response.Content.ReadAsStringAsync();
var documents = JsonSerializer.Deserialize<List<Dictionary<string, object>>>(json);
var searchClient = new SearchClient(new Uri(searchEndpoint), indexName, credential);
var searchIndexingBufferedSender = new
SearchIndexingBufferedSender<Dictionary<string, object>>(
    searchClient,
    new SearchIndexingBufferedSenderOptions<Dictionary<string, object>>
    {
        KeyFieldAccessor = doc => doc["id"].ToString(),
    }
);
await searchIndexingBufferedSender.UploadDocumentsAsync(documents);
await searchIndexingBufferedSender.FlushAsync();
Console.WriteLine($"Documents uploaded to index '{indexName}' successfully.");
```

Create a knowledge source

A knowledge source is a reusable reference to source data. The following code defines a knowledge source named `earth-knowledge-source` that targets the `earth-at-night` index.

`SourceDataFields` specifies which index fields are accessible for retrieval and citations. Our example includes only human-readable fields to avoid lengthy, uninterpretable embeddings in responses.

C#

```
// Create a knowledge source
var indexKnowledgeSource = new SearchIndexKnowledgeSource(
    name: knowledgeSourceName,
    searchIndexParameters: new SearchIndexKnowledgeSourceParameters(searchIndexName:
indexName)
{
    SourceDataFields = { new SearchIndexFieldReference(name: "id"), new
SearchIndexFieldReference(name: "page_chunk"), new SearchIndexFieldReference(name:
"page_number") }
}
);

await indexClient.CreateOrUpdateKnowledgeSourceAsync(indexKnowledgeSource);
```

```
Console.WriteLine($"Knowledge source '{knowledgeSourceName}' created or updated successfully.");
```

Create a knowledge base

To target `earth-knowledge-source` and your `gpt-5-mini` deployment at query time, you need a knowledge base. The following code defines a knowledge base named `earth-knowledge-base`, which you previously specified using the `knowledgeBaseName` variable.

`OutputMode` is set to `AnswerSynthesis`, enabling natural-language answers that cite the retrieved documents and follow the provided `AnswerInstructions`.

C#

```
// Create a knowledge base
var openAiParameters = new AzureOpenAIVectorizerParameters
{
    ResourceUri = new Uri(aaaiEndpoint),
    DeploymentName = aaaiGptDeployment,
    ModelName = aaaiGptModel
};

var model = new KnowledgeBaseAzureOpenAIModel(azureOpenAIParameters:
openAiParameters);

var knowledgeBase = new KnowledgeBase(
    name: knowledgeBaseName,
    knowledgeSources: new KnowledgeSourceReference[] { new
KnowledgeSourceReference(knowledgeSourceName) }
)
{
    RetrievalReasoningEffort = new KnowledgeRetrievalLowReasoningEffort(),
    OutputMode = KnowledgeRetrievalOutputMode.AnswerSynthesis,
    AnswerInstructions = "Provide a two sentence concise and informative answer
based on the retrieved documents.",
    Models = { model }
};

await indexClient.CreateOrUpdateKnowledgeBaseAsync(knowledgeBase);
Console.WriteLine($"Knowledge base '{knowledgeBaseName}' created or updated
successfully.");
```

Set up messages

Messages are the input for the retrieval route and contain the conversation history. Each message includes a role that indicates its origin, such as `system` or `user`, and content in natural language. The LLM you use determines which roles are valid.

The following code creates a system message, which instructs `earth-knowledge-base` to answer questions about the Earth at night and respond with "I don't know" when answers are unavailable.

C#

```
// Set up messages
string instructions = @"A Q&A agent that can answer questions about the Earth at
night.
If you don't have the answer, respond with ""I don't know"".";

var messages = new List<Dictionary<string, string>>
{
    new Dictionary<string, string>
    {
        { "role", "system" },
        { "content", instructions }
    }
};
```

Run the retrieval pipeline

You're ready to run agentic retrieval. The following code sends a two-part user query to `earth-knowledge-base`, which:

1. Analyzes the entire conversation to infer the user's information need.
2. Decomposes the compound query into focused subqueries.
3. Runs the subqueries concurrently against your knowledge source.
4. Uses semantic ranker to rerank and filter the results.
5. Synthesizes the top results into a natural-language answer.

C#

```
// Run agentic retrieval
var baseClient = new KnowledgeBaseRetrievalClient(
    endpoint: new Uri(searchEndpoint),
    knowledgeBaseName: knowledgeBaseName,
    tokenCredential: new DefaultAzureCredential()
);

messages.Add(new Dictionary<string, string>
{
    { "role", "user" },
    { "content", @"Why do suburban belts display larger December brightening than
urban cores even though absolute light levels are higher downtown? Why is the
Phoenix nighttime street grid is so sharply visible from space, whereas large
stretches of the interstate between midwestern cities remain comparatively dim?" }
});
```

```

var retrievalRequest = new KnowledgeBaseRetrievalRequest();
foreach (Dictionary<string, string> message in messages) {
    if (message["role"] != "system") {
        retrievalRequest.Messages.Add(new KnowledgeBaseMessage(content: new[] { new KnowledgeBaseMessageTextContent(message["content"]) }) { Role = message["role"] });
    }
}
retrievalRequest.RetrievalReasoningEffort = new KnowledgeRetrievalLowReasoningEffort();
var retrievalResult = await baseClient.RetrieveAsync(retrievalRequest);

messages.Add(new Dictionary<string, string>
{
    { "role", "assistant" },
    { "content", (retrievalResult.Value.Response[0].Content[0] as KnowledgeBaseMessageTextContent).Text }
});

```

Review the response, activity, and references

The following code displays the response, activity, and references from the retrieval pipeline, where:

- `Response` provides a synthesized, LLM-generated answer to the query that cites the retrieved documents. When answer synthesis isn't enabled, this section contains content extracted directly from the documents.
- `Activity` tracks the steps that were taken during the retrieval process, including the subqueries generated by your `gpt-5-mini` deployment and the tokens used for semantic ranking, query planning, and answer synthesis.
- `References` lists the documents that contributed to the response, each one identified by their `DocKey`.

C#

```

// Print the response, activity, and references
Console.WriteLine("Response:");
Console.WriteLine((retrievalResult.Value.Response[0].Content[0] as KnowledgeBaseMessageTextContent).Text);

Console.WriteLine("Activity:");
foreach (var activity in retrievalResult.Value.Activity)
{
    Console.WriteLine($"Activity Type: {activity.GetType().Name}");
    string activityJson = JsonSerializer.Serialize(
        activity,
        activity.GetType(),
        new JsonSerializerOptions { WriteIndented = true })

```

```

    );
    Console.WriteLine(activityJson);
}

Console.WriteLine("References:");
foreach (var reference in retrievalResult.Value.References)
{
    Console.WriteLine($"Reference Type: {reference.GetType().Name}");
    string referenceJson = JsonSerializer.Serialize(
        reference,
        reference.GetType(),
        new JsonSerializerOptions { WriteIndented = true }
    );
    Console.WriteLine(referenceJson);
}

```

Continue the conversation

The following code continues the conversation with `earth-knowledge-base`. After you send this user query, the knowledge base fetches relevant content from `earth-knowledge-source` and appends the response to the messages list.

C#

```

// Continue the conversation
messages.Add(new Dictionary<string, string>
{
    { "role", "user" },
    { "content", "How do I find lava at night?" }
});

retrievalRequest = new KnowledgeBaseRetrievalRequest();
foreach (Dictionary<string, string> message in messages) {
    if (message["role"] != "system") {
        retrievalRequest.Messages.Add(new KnowledgeBaseMessage(content: new[] { new
KnowledgeBaseMessageTextContent(message["content"]) }) { Role = message["role"] });
    }
}
retrievalRequest.RetrievalReasoningEffort = new
KnowledgeRetrievalLowReasoningEffort();
retrievalResult = await baseClient.RetrieveAsync(retrievalRequest);

messages.Add(new Dictionary<string, string>
{
    { "role", "assistant" },
    { "content", (retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent).Text }
});

```

Review the new response, activity, and references

The following code displays the new response, activity, and references from the retrieval pipeline.

C#

```
// Print the new response, activity, and references
Console.WriteLine("Response:");
Console.WriteLine((retrievalResult.Value.Response[0].Content[0] as
KnowledgeBaseMessageTextContent).Text);

Console.WriteLine("Activity:");
foreach (var activity in retrievalResult.Value.Activity)
{
    Console.WriteLine($"Activity Type: {activity.GetType().Name}");
    string activityJson = JsonSerializer.Serialize(
        activity,
        activity.GetType(),
        new JsonSerializerOptions { WriteIndented = true }
    );
    Console.WriteLine(activityJson);
}

Console.WriteLine("References:");
foreach (var reference in retrievalResult.Value.References)
{
    Console.WriteLine($"Reference Type: {reference.GetType().Name}");
    string referenceJson = JsonSerializer.Serialize(
        reference,
        reference.GetType(),
        new JsonSerializerOptions { WriteIndented = true }
    );
    Console.WriteLine(referenceJson);
}
```

Clean up resources

When you work in your own subscription, it's a good idea to finish a project by determining whether you still need the resources you created. Resources that are left running can cost you money.

In the Azure portal, you can manage your Azure AI Search and Foundry resources by selecting **All resources** or **Resource groups** from the left pane.

Otherwise, the following code from `Program.cs` deleted the objects you created in this quickstart.

Delete the knowledge base

C#

```
await indexClient.DeleteKnowledgeBaseAsync(knowledgeBaseName);
Console.WriteLine($"Knowledge base '{knowledgeBaseName}' deleted successfully.");
```

Delete the knowledge source

C#

```
await indexClient.DeleteKnowledgeSourceAsync(knowledgeSourceName);
Console.WriteLine($"Knowledge source '{knowledgeSourceName}' deleted
successfully.");
```

Delete the search index

C#

```
await indexClient.DeleteIndexAsync(indexName);
Console.WriteLine($"Index '{indexName}' deleted successfully.");
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Create a knowledge base](#)
- [Use a knowledge base to retrieve data](#)
- [Tutorial: Build an end-to-end agentic retrieval solution](#)

Last updated on 11/18/2025

Quickstart: Vector search

In this quickstart, you work with a .NET app to create, populate, and query vectors. The code examples perform these operations by using the [Azure AI Search client library](#). The library provides an abstraction over the REST API for access to index operations such as data ingestion, search operations, and index management operations.

In Azure AI Search, a [vector store](#) has an index schema that defines vector and nonvector fields, a vector search configuration for algorithms that create the embedding space, and settings on vector field definitions that are evaluated at query time. The [Create Index](#) REST API creates the vector store.

(!) Note

This quickstart omits the vectorization step and provides inline embeddings. If you want to add [built-in data chunking and vectorization](#) over your own content, try the [Import data \(new\) wizard](#) for an end-to-end walkthrough.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription.
 - You can use a free search service for most of this quickstart, but we recommend the Basic tier or higher for larger data files.
 - To run the query example that invokes [semantic reranking](#), your search service must have [semantic ranker enabled](#).
- [Visual Studio Code](#) or [Visual Studio](#).
- [Git](#) to clone the sample repo.

Get service information

Requests to the search endpoint must be authenticated and authorized. While it's possible to use API keys or roles for this task, we recommend [using a keyless connection via Microsoft Entra ID](#).

This quickstart uses `DefaultAzureCredential`, which simplifies authentication in both development and production scenarios. However, for production scenarios, you might have

more advanced requirements that require a different approach. See [Authenticate .NET apps to Azure services by using the Azure SDK for .NET](#) to understand all of your options.

Clone the code and setup environment

1. Clone the repo containing the code for this quickstart.

```
Bash
```

```
git clone https://github.com/Azure-Samples/azure-search-dotnet-samples
```

This repo has .NET code examples for several articles each in a separate subfolder.

2. Open the subfolder `quickstart-Vector-Search` in Visual Studio Code, or double click the `.sln` file to open the solution in Visual Studio.
3. Open the `appsettings.json` files in the `VectorSearchExamples` and `VectorSearchCreatePopulateIndex` folders. Update the following values:

- `AZURE_SEARCH_ENDPOINT`: Find the url of your Azure AI Search service in the [Azure portal](#). On the **Overview** page of your search resource, look for the URL field. An example endpoint might look like `https://mydemo.search.windows.net`.
- `AZURE_SEARCH_INDEX_NAME`: Leave the default value provided in the file or enter your own index name.

Create the vector index and upload documents

To run search queries against the Azure AI Search service, you first need to create a search index and upload documents to the service.

1. Open a new terminal in the `VectorSearchCreatePopulateIndex` folder.
2. Run the project using the `dotnet run` command:

```
.NET CLI
```

```
dotnet run
```

The following code executes to create an index:

```
C#
```

```

static async Task CreateSearchIndex(string indexName, SearchIndexClient indexClient)
{
    var addressField = new ComplexField("Address");
    addressField.Fields.Add(new SearchableField("StreetAddress") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft });
    addressField.Fields.Add(new SearchableField("City") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true });
    addressField.Fields.Add(new SearchableField("StateProvince") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true });
    addressField.Fields.Add(new SearchableField("PostalCode") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true });
    addressField.Fields.Add(new SearchableField("Country") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true });

    var allFields = new List<SearchField>()
{
    new SimpleField("HotelId", SearchFieldDataType.String) { IsKey = true,
IsFacetable = true, IsFilterable = true },
    new SearchableField("HotelName") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft },
    new SearchableField("Description") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft },
    new VectorSearchField("DescriptionVector", 1536, "my-vector-profile"),
    new SearchableField("Category") { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true },
    new SearchableField("Tags", collection: true) { AnalyzerName =
LexicalAnalyzerName.EnMicrosoft, IsFacetable = true, IsFilterable = true },
    new SimpleField("ParkingIncluded", SearchFieldDataType.Boolean) { IsFacetable =
true, IsFilterable = true },
    new SimpleField("LastRenovationDate", SearchFieldDataType.DateTimeOffset) {
IsSortable = true },
    new SimpleField("Rating", SearchFieldDataType.Double) { IsFacetable = true,
IsFilterable = true, IsSortable = true },
    addressField,
    new SimpleField("Location", SearchFieldDataType.GeographyPoint) { IsFilterable =
true, IsSortable = true },
};
}

// Create the suggester configuration
var suggester = new SearchSuggester("sg", new[] { "Address/City",
"Address/Country" });

// Create the semantic search
var semanticSearch = new SemanticSearch()
{
    Configurations =
    {
        new SemanticConfiguration(
            name: "semantic-config",
            prioritizedFields: new SemanticPrioritizedFields
            {
                TitleField = new SemanticField("HotelName"),
                KeywordsFields = { new SemanticField("Category") },
                ContentFields = { new SemanticField("Description") }
            }
    }
}

```

```

        })
    };
};

// Add vector search configuration
var vectorSearch = new VectorSearch();
vectorSearch.Algorithms.Add(new HnswAlgorithmConfiguration(name: "my-hnsw-
vector-config-1"));
vectorSearch.Profiles.Add(new VectorSearchProfile(name: "my-vector-profile",
algorithmConfigurationName: "my-hnsw-vector-config-1"));

var definition = new SearchIndex(indexName)
{
    Fields = allFields,
    Suggesters = { suggester },
    VectorSearch = vectorSearch,
    SemanticSearch = semanticSearch
};

// Create or update the index
Console.WriteLine($"Creating or updating index '{indexName}'...");
var result = await indexClient.CreateOrUpdateIndexAsync(definition);
Console.WriteLine($"Index '{result.Value.Name}' updated.");
Console.WriteLine();
}

```

The following code uploads the JSON formatted documents in the `hotel-samples.json` file to the Azure AI Search service:

C#

```

static async Task UploadDocs(SearchClient searchClient)
{
    var jsonPath = Path.Combine(Directory.GetCurrentDirectory(), "HotelData.json");

    // Read and parse hotel data
    var json = await File.ReadAllTextAsync(jsonPath);
    List<Hotel> hotels = new List<Hotel>();
    try
    {
        using var doc = JsonDocument.Parse(json);
        if (doc.RootElement.ValueKind != JsonValueKind.Array)
        {
            Console.WriteLine("HotelData.json root is not a JSON array.");
        }
        // Deserialize all hotel objects
        hotels = doc.RootElement.EnumerateArray()
            .Select(e => JsonSerializer.Deserialize<Hotel>(e.GetRawText()))
            .Where(h => h != null)
            .ToList();
    }
    catch (JsonException ex)
    {

```

```

        Console.WriteLine($"Failed to parse HotelData.json: {ex.Message}");
    }

    try
    {
        // Upload hotel documents to Azure Search
        var result = await searchClient.UploadDocumentsAsync(hotels);
        foreach (var r in result.Value.Results)
        {
            Console.WriteLine($"Key: {r.Key}, Succeeded: {r.Succeeded}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed to upload documents: " + ex);
    }
}

```

After you run the project, the following output is printed:

Output

```

Key: 1, Succeeded: True
Key: 2, Succeeded: True
Key: 3, Succeeded: True
Key: 4, Succeeded: True
Key: 48, Succeeded: True
Key: 49, Succeeded: True
Key: 13, Succeeded: True

```

Key takeaways:

- Your code interacts with a specific search index hosted in your Azure AI Search service through the `SearchClient`, which is the main object provided by the `azure-search-documents` package. The `SearchClient` provides access to index operations such as:
 - **Data ingestion:** `UploadDocuments()`, `MergeDocuments()`, `DeleteDocuments()`
 - **Search operations:** `Search()`, `Autocomplete()`, `Suggest()`
 - **Index management operations:** `CreateOrUpdateIndex()`
- Vector fields contain floating point values. The dimensions attribute has a minimum of 2 and a maximum of 4096 floating point values each. This size of embeddings generated by the Azure OpenAI `text-embedding-3-small` model for this quickstart is 1536.

Run search queries

After the index is created and documents are loaded, you can issue vector queries against them by calling `SearchAsync()` with various parameters.

1. In the `VectorSearchExamples` folder, open the `Program.cs` file.

2. Open a new terminal in the `VectorSearchExamples` folder.

In the following sections, you run queries against the `hotels-vector-quickstart` index. The queries include:

- Single vector search
- Single vector search with filter
- Hybrid search
- Semantic hybrid search with filter

Single vector search

The first example demonstrates a basic scenario where you want to find document descriptions that closely match the search string.

1. In the `Program.cs` file of the `VectorSearchExamples` folder, uncomment the method call `SearchExamples.SearchSingleVector(searchClient, vectorizedResult);`. This method executes the following search function in the `SearchExamples.cs` class:

C#

```
public static async Task SearchSingleVector(SearchClient searchClient,
    ReadOnlyMemory<float> precalculatedVector)
{
    SearchResults<Hotel> response = await searchClient.SearchAsync<Hotel>(
        new SearchOptions
        {
            VectorSearch = new()
            {
                Queries = { new VectorizedQuery(precalculatedVector) {
                    KNearestNeighborsCount = 5, Fields = { "DescriptionVector" } } },
                Select = { "HotelId", "HotelName", "Description", "Category",
                    "Tags" },
            });
}

Console.WriteLine($"Single Vector Search Results:");
await foreach (SearchResult<Hotel> result in response.GetResultsAsync())
{
    Hotel doc = result.Document;
    Console.WriteLine($"Score: {result.Score}, HotelId: {doc.HotelId},
        HotelName: {doc.HotelName}");
}
Console.WriteLine();
}
```

2. Run the project using the `dotnet run` command:

```
.NET CLI
```

```
dotnet run
```

After you run the project, the search results are printed in the output window:

```
Output
```

```
Single Vector Search Results:  
Score: 0.6605852, HotelId: 48, HotelName: Nordick's Valley Motel  
Score: 0.6333684, HotelId: 13, HotelName: Luxury Lion Resort  
Score: 0.605672, HotelId: 4, HotelName: Sublime Palace Hotel  
Score: 0.6026341, HotelId: 49, HotelName: Swirling Currents Hotel  
Score: 0.57902366, HotelId: 2, HotelName: Old Century Hotel
```

Single vector search with filter

You can add filters, but the filters are applied to the nonvector content in your index. In this example, the filter applies to the `Tags` field to filter out any hotels that don't provide free Wi-Fi.

1. In the `Program.cs` file of the `VectorSearchExamples` folder, uncomment the method call `SearchExamples.SearchSingleVectorWithFilter(searchClient, vectorizedResult);`. This method executes the following search function in the `SearchExamples.cs` class:

```
C#
```

```
public static async Task SearchSingleVectorWithFilter(SearchClient searchClient, ReadOnlyMemory<float> precalculatedVector)
{
    SearchResults<Hotel> responseWithFilter = await
    searchClient.SearchAsync<Hotel>(
        new SearchOptions
        {
            VectorSearch = new()
            {
                Queries = { new VectorizedQuery(precalculatedVector) {
                    KNearestNeighborsCount = 5, Fields = { "DescriptionVector" } } }
            },
            Filter = "Tags/any(tag: tag eq 'free wifi')",
            Select = { "HotelId", "HotelName", "Description", "Category",
"Tags" }
        });
    Console.WriteLine($"Single Vector Search With Filter Results:");
    await foreach (SearchResult<Hotel> result in
    responseWithFilter.GetResultsAsync())
    {
        Hotel doc = result.Document;
        Console.WriteLine($"Score: {result.Score}, HotelId: {doc.HotelId},
```

```

        HotelName: {doc.HotelName}, Tags: {string.Join(String.Empty, doc.Tags)}");
    }
    Console.WriteLine();
}

```

2. Run the project again, and the status of each document is printed below it:

Output

```

Single Vector Search With Filter Results:
Score: 0.6605852, HotelId: 48, HotelName: Nordick's Valley Motel, Tags:
continental breakfastair conditioningfree wifi
Score: 0.57902366, HotelId: 2, HotelName: Old Century Hotel, Tags: poolfree
wifiair conditioningconcierge

```

The query was the same as the previous [single vector search example](#), but it includes a post-processing exclusion filter and returns only the two hotels that have free Wi-Fi.

3. The next filter example uses a **geo filter**. In the `Program.cs` file of the

`VectorSearchExamples` folder, uncomment the method call

`SearchExamples.SingleSearchWithGeoFilter(searchClient, vectorizedResult);`. This method executes the following search function in the `SearchExamples.cs` class:

C#

```

public static async Task SingleSearchWithGeoFilter(SearchClient searchClient,
    ReadOnlyMemory<float> precalculatedVector)
{
    SearchResults<Hotel> responseWithGeoFilter = await
    searchClient.SearchAsync<Hotel>(
        new SearchOptions
        {
            VectorSearch = new()
            {
                Queries = { new VectorizedQuery(precalculatedVector) {
                    KNearestNeighborsCount = 5, Fields = { "DescriptionVector" } } }
            },
            Filter = "geo.distance(Location, geography'POINT(-77.03241
38.90166') le 300",
            Select = { "HotelId", "HotelName", "Description", "Address",
"Category", "Tags" },
            Facets = { "Address/StateProvince" },
        });
}

Console.WriteLine($"Vector query with a geo filter:");
await foreach (SearchResult<Hotel> result in
responseWithGeoFilter.GetResultsAsync())
{
    Hotel doc = result.Document;
    Console.WriteLine($"HotelId: {doc.HotelId}");
}

```

```

        Console.WriteLine($"HotelName: {doc.HotelName}");
        Console.WriteLine($"Score: {result.Score}");
        Console.WriteLine($"City/State:
{doc.Address.City}/{doc.Address.StateProvince}");
        Console.WriteLine($"Description: {doc.Description}");
        Console.WriteLine();
    }
    Console.WriteLine();
}

```

The query was the same as the previous [single vector search example](#), but it includes a post-processing exclusion filter and returns only the two hotels within 300 kilometers.

- Run the project again, and the status of each document is printed below it:

Output

```

Vector query with a geo filter:
-HotelId: 48
HotelName: Nordick's Valley Motel
Score: 0.6605852246284485
City/State: Washington D.C./null
Description: Only 90 miles (about 2 hours) from the nation's capital and nearby
most everything the historic valley has to offer. Hiking? Wine Tasting?
Exploring the caverns? It's all nearby and we have specially priced packages to
help make our B&B your home base for fun while visiting the valley.

-HotelId: 49
HotelName: Swirling Currents Hotel
Score: 0.602634072303772
City/State: Arlington/VA
Description: Spacious rooms, glamorous suites and residences, rooftop pool,
walking access to shopping, dining, entertainment and the city center. Each
room comes equipped with a microwave, a coffee maker and a minifridge. In-room
entertainment includes complimentary Wi-Fi and flat-screen TVs.

```

Hybrid search

Hybrid search consists of keyword queries and vector queries in a single search request. This example runs the vector query and full-text search concurrently:

- Search string:** `historic hotel walk to restaurants and shopping`
- Vector query string** (vectorized into a mathematical representation): `Quintessential
lodging near running trails, eateries, retail`

- In the `Program.cs` file of the `VectorSearchExamples` folder, uncomment the method call `SearchExamples.SearchHybridVectorAndText(searchClient, vectorizedResult);`. This method executes the following search function in the `SearchExamples.cs` class:

C#

```
public static async Task<SearchResults<Hotel>>
SearchHybridVectorAndText(SearchClient searchClient, ReadOnlyMemory<float>
precalculatedVector)
{
    SearchResults<Hotel> responseWithFilter = await
searchClient.SearchAsync<Hotel>(
        "historic hotel walk to restaurants and shopping",
        new SearchOptions
    {
        VectorSearch = new()
        {
            Queries = { new VectorizedQuery(precalculatedVector) {
KNearestNeighborsCount = 5, Fields = { "DescriptionVector" } } }
        },
        Select = { "HotelId", "HotelName", "Description", "Category",
"Tags" },
        Size = 5,
    });
}

Console.WriteLine($"Hybrid search results:");
await foreach (SearchResult<Hotel> result in
responseWithFilter.GetResultsAsync())
{
    Hotel doc = result.Document;
    Console.WriteLine($"Score: {result.Score}");
    Console.WriteLine($"HotelId: {doc.HotelId}");
    Console.WriteLine($"HotelName: {doc.HotelName}");
    Console.WriteLine($"Description: {doc.Description}");
    Console.WriteLine($"Category: {doc.Category}");
    Console.WriteLine($"Tags: {string.Join(String.Empty, doc.Tags)}");
    Console.WriteLine();
}
Console.WriteLine();
return responseWithFilter;
}
```

2. Run the project again, and the status of each document is printed below it:

Output

```
Hybrid search results:
Score: 0.03279569745063782
HotelId: 4
HotelName: Sublime Palace Hotel
Description: Sublime Palace Hotel is located in the heart of the historic
center of Sublime in an extremely vibrant and lively area within short walking
distance to the sites and landmarks of the city and is surrounded by the
extraordinary beauty of churches, buildings, shops and monuments. Sublime Cliff
is part of a lovingly restored 19th century resort, updated for every modern
convenience.
Category: Boutique
```

Tags: conciergeviewair conditioning

Score: 0.032786883413791656

HotelId: 13

HotelName: Luxury Lion Resort

Description: Unmatched Luxury. Visit our downtown hotel to indulge in luxury accommodations. Moments from the stadium and transportation hubs, we feature the best in convenience and comfort.

Category: Luxury

Tags: barconciergerestaurant

Score: 0.03205128386616707

HotelId: 48

HotelName: Nordick's Valley Motel

Description: Only 90 miles (about 2 hours) from the nation's capital and nearby most everything the historic valley has to offer. Hiking? Wine Tasting? Exploring the caverns? It's all nearby and we have specially priced packages to help make our B&B your home base for fun while visiting the valley.

Category: Boutique

Tags: continental breakfastair conditioningfree wifi

Score: 0.0317460335791111

HotelId: 49

HotelName: Swirling Currents Hotel

Description: Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center. Each room comes equipped with a microwave, a coffee maker and a minifridge. In-room entertainment includes complimentary W-Fi and flat-screen TVs.

Category: Suite

Tags: air conditioninglaundry service24-hour front desk service

Score: 0.03077651560306549

HotelId: 2

HotelName: Old Century Hotel

Description: The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.

Category: Boutique

Tags: poolfree wifiair conditioningconcierge

Because Reciprocal Rank Fusion (RRF) merges results, it helps to review the inputs. The following results are from only the full-text query. The top two results are Sublime Palace Hotel and History Lion Resort. The Sublime Palace Hotel has a stronger BM25 relevance score.

JSON

```
{  
  "@search.score": 2.2626662,  
  "HotelName": "Sublime Palace Hotel",  
  "Description": "Sublime Palace Hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking
```

```

distance to the sites and landmarks of the city and is surrounded by the
extraordinary beauty of churches, buildings, shops and monuments. Sublime Palace is
part of a lovingly restored 1800 palace."
},
{
    "@search.score": 0.86421645,
    "HotelName": "Luxury Lion Resort",
    "Description": "Unmatched Luxury. Visit our downtown hotel to indulge in luxury
accommodations. Moments from the stadium, we feature the best in comfort"
},

```

In the vector-only query, which uses HNSW for finding matches, the Sublime Palace Hotel drops to fourth position. Historic Lion, which was second in the full-text search and third in the vector search, doesn't experience the same range of fluctuation, so it appears as a top match in a homogenized result set.

JSON

```

"value": [
{
    "@search.score": 0.857736,
    "HotelId": "48",
    "HotelName": "Nordick's Valley Motel",
    "Description": "Only 90 miles (about 2 hours) from the nation's capital and
nearby most everything the historic valley has to offer. Hiking? Wine Tasting?
Exploring the caverns? It's all nearby and we have specially priced packages to
help make our B&B your home base for fun while visiting the valley.",
    "Category": "Boutique"
},
{
    "@search.score": 0.8399129,
    "HotelId": "49",
    "HotelName": "Swirling Currents Hotel",
    "Description": "Spacious rooms, glamorous suites and residences, rooftop
pool, walking access to shopping, dining, entertainment and the city center.",
    "Category": "Luxury"
},
{
    "@search.score": 0.8383954,
    "HotelId": "13",
    "HotelName": "Luxury Lion Resort",
    "Description": "Unmatched Luxury. Visit our downtown hotel to indulge in
luxury accommodations. Moments from the stadium, we feature the best in comfort",
    "Category": "Resort and Spa"
},
{
    "@search.score": 0.8254346,
    "HotelId": "4",
    "HotelName": "Sublime Palace Hotel",
    "Description": "Sublime Palace Hotel is located in the heart of the historic
center of Sublime in an extremely vibrant and lively area within short walking
distance to the sites and landmarks of the city and is surrounded by the
extraordinary beauty of churches, buildings, shops and monuments. Sublime Palace is

```

```

part of a lovingly restored 1800 palace.",  

    "Category": "Boutique"  

},  

{  

    "@search.score": 0.82380056,  

    "HotelId": "1",  

    "HotelName": "Stay-Kay City Hotel",  

    "Description": "The hotel is ideally located on the main commercial artery  

of the city in the heart of New York.",  

    "Category": "Boutique"  

},  

{  

    "@search.score": 0.81514084,  

    "HotelId": "2",  

    "HotelName": "Old Century Hotel",  

    "Description": "The hotel is situated in a nineteenth century plaza, which  

has been expanded and renovated to the highest architectural standards to create a  

modern, functional and first-class hotel in which art and unique historical elements  

coexist with the most modern comforts.",  

    "Category": "Boutique"  

},  

{  

    "@search.score": 0.8133763,  

    "HotelId": "3",  

    "HotelName": "Gastronomic Landscape Hotel",  

    "Description": "The Hotel stands out for its gastronomic excellence under  

the management of William Dough, who advises on and oversees all of the Hotel's  

restaurant services.",  

    "Category": "Resort and Spa"  

}  

]

```

Semantic hybrid search with a filter

The hybrid query with semantic ranking is filtered to show only the hotels within a 500-kilometer radius of Washington D.C.

1. In the `Program.cs` file of the `VectorSearchExamples` folder, uncomment the method call `SearchExamples.SearchHybridVectorAndSemantic(searchClient, vectorizedResult);`. This method executes the following search function in the `SearchExamples.cs` class:

C#

```

public static async Task SearchHybridVectorAndSemantic(SearchClient  

searchClient, ReadOnlyMemory<float> precalculatedVector)  

{  

    SearchResults<Hotel> responseWithFilter = await  

searchClient.SearchAsync<Hotel>(  

    "historic hotel walk to restaurants and shopping",  

    new SearchOptions  

{

```

```

        IncludeTotalCount = true,
        VectorSearch = new()
        {
            Queries = { new VectorizedQuery(precalculatedVector) {
                KNearestNeighborsCount = 5, Fields = { "DescriptionVector" } } }
            },
            Select = { "HotelId", "HotelName", "Description", "Category",
"Tags" },
            SemanticSearch = new SemanticSearchOptions
            {
                SemanticConfigurationName = "semantic-config"
            },
            QueryType = SearchQueryType.Semantic,
            Size = 5
        });
    }

    Console.WriteLine($"Hybrid search results:");
    await foreach (SearchResult<Hotel> result in
responseWithFilter.GetResultsAsync())
    {
        Hotel doc = result.Document;
        Console.WriteLine($"Score: {result.Score}");
        Console.WriteLine($"HotelId: {doc.HotelId}");
        Console.WriteLine($"HotelName: {doc.HotelName}");
        Console.WriteLine($"Description: {doc.Description}");
        Console.WriteLine($"Category: {doc.Category}");
        Console.WriteLine();
    }
    Console.WriteLine();
}

```

- Run the project again, and review the output below the cell. The response is three hotels, which are filtered by location and faceted by `StateProvince` and semantically reranked to promote results that are closest to the search string query (`historic hotel walk to restaurants and shopping`).

The Swirling Currents Hotel now moves into the top spot. Without semantic ranking, Nordick's Valley Motel is number one. With semantic ranking, the machine comprehension models recognize that `historic` applies to "hotel, within walking distance to dining (restaurants) and shopping."

Output

```

Total semantic hybrid results: 7
- Score: 0.0317460335791111
  Re-ranker Score: 2.6550590991973877
  HotelId: 49
  HotelName: Swirling Currents Hotel
  Description: Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center. Each room comes equipped with a microwave, a coffee maker and a minifridge. In-room

```

entertainment includes complimentary Wi-Fi and flat-screen TVs.

Category: Suite

- Score: 0.03279569745063782

Re-ranker Score: 2.599761724472046

HotelId: 4

HotelName: Sublime Palace Hotel

Description: Sublime Palace Hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking distance to the sites and landmarks of the city and is surrounded by the extraordinary beauty of churches, buildings, shops and monuments. Sublime Cliff is part of a lovingly restored 19th century resort, updated for every modern convenience.

Category: Boutique

- Score: 0.03125

Re-ranker Score: 2.3480887413024902

HotelId: 2

HotelName: Old Century Hotel

Description: The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.

Category: Boutique

- Score: 0.016393441706895828

Re-ranker Score: 2.2718777656555176

HotelId: 1

HotelName: Stay-Kay City Hotel

Description: This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic center of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.

Category: Boutique

- Score: 0.01515151560306549

Re-ranker Score: 2.0582215785980225

HotelId: 3

HotelName: Gastronomic Landscape Hotel

Description: The Gastronomic Hotel stands out for its culinary excellence under the management of William Dough, who advises on and oversees all of the Hotel's restaurant services.

Category: Suite

Key takeaways:

- In a hybrid search, you can integrate vector search with full-text search over keywords. Filters, spell check, and semantic ranking apply to textual content only, and not vectors. In this final query, there's no semantic `answer` because the system didn't produce one that was sufficiently strong.
- Actual results include more detail, including semantic captions and highlights. Results were modified for readability. To get the full structure of the response, run the request in the REST client.

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal by using the [All resources](#) or [Resource groups](#) link in the leftmost pane.

Next steps

- Review the repository of code samples for vector search capabilities in Azure AI Search for [.NET](#)
 - Review the other .NET and Azure AI Search code samples in the [azure-search-dotnet-samples repo](#)
-

Last updated on 11/20/2025

Quickstart: Classic generative search (RAG) using grounding data from Azure AI Search

10/15/2025

In this quickstart, you send queries to a chat completion model for a conversational search experience over your indexed content on Azure AI Search. After setting up Azure OpenAI and Azure AI Search resources in the Azure portal, you run code to call the APIs.

ⓘ Note

We now recommend [agentic retrieval](#) for RAG workflows, but classic RAG is simpler. If it meets your application requirements, it's still a good choice.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure OpenAI resource](#).
 - [Choose a region](#) that supports the chat completion model you want to use (gpt-4o, gpt-4o-mini, or an equivalent model).
 - [Deploy the chat completion model](#) in Azure AI Foundry or [use another approach](#).
- An [Azure AI Search resource](#).
 - We recommend using the Basic tier or higher.
 - [Enable semantic ranking](#).
- [Visual Studio Code](#) or [Visual Studio](#).
- [.NET 9.0](#) installed.

Configure access

Requests to the search endpoint must be authenticated and authorized. You can use API keys or roles for this task. Keys are easier to start with, but roles are more secure. This quickstart assumes roles.

You're setting up two clients, so you need permissions on both resources.

Azure AI Search is receiving the query request from your local system. Assign yourself the **Search Index Data Reader** role assignment if the hotels sample index already exists. If it

doesn't exist, assign yourself **Search Service Contributor** and **Search Index Data Contributor** roles so that you can create and query the index.

Azure OpenAI is receiving the query and the search results from your local system. Assign yourself the **Cognitive Services OpenAI User** role on Azure OpenAI.

1. Sign in to the [Azure portal](#).
2. Configure Azure AI Search for role-based access:
 - a. In the Azure portal, find your Azure AI Search service.
 - b. On the left menu, select **Settings > Keys**, and then select either **Role-based access control** or **Both**.
3. Assign roles:
 - a. On the left menu, select **Access control (IAM)**.
 - b. On Azure AI Search, select these roles to create, load, and query a search index, and then assign them to your Microsoft Entra ID user identity:
 - **Search Index Data Contributor**
 - **Search Service Contributor**
 - c. On Azure OpenAI, select **Access control (IAM)** to assign this role to yourself on Azure OpenAI:
 - **Cognitive Services OpenAI User**

It can take several minutes for permissions to take effect.

Create an index

A search index provides grounding data for the chat model. We recommend the **hotels-sample-index**, which can be created in minutes and runs on any search service tier. This index is created using built-in sample data.

1. In the Azure portal, [find your search service](#).
2. On the **Overview** home page, select **Import data** to start the wizard.
3. On the **Connect to your data** page, select **Samples** from the dropdown list.
4. Choose the **hotels-sample**.
5. Select **Next** through the remaining pages, accepting the default values.

6. Once the index is created, select **Search management > Indexes** from the left menu to open the index.
7. Select **Edit JSON**.
8. Scroll to the end of the index, where you can find placeholders for constructs that can be added to an index.

JSON

```
"analyzers": [],
"tokenizers": [],
"tokenFilters": [],
"charFilters": [],
"normalizers": [],
```

9. On a new line after "normalizers", paste in the following semantic configuration. This example specifies a `"defaultConfiguration"`, which is important to the running of this quickstart.

JSON

```
"semantic": {
    "defaultConfiguration": "semantic-config",
    "configurations": [
        {
            "name": "semantic-config",
            "prioritizedFields": {
                "titleField": {
                    "fieldName": "HotelName"
                },
                "prioritizedContentFields": [
                    {
                        "fieldName": "Description"
                    }
                ],
                "prioritizedKeywordsFields": [
                    {
                        "fieldName": "Category"
                    },
                    {
                        "fieldName": "Tags"
                    }
                ]
            }
        }
    ],
}
```

10. Save your changes.

11. Run the following query in [Search Explorer](#) to test your index: complimentary breakfast.

Output should look similar to the following example. Results that are returned directly from the search engine consist of fields and their verbatim values, along with metadata like a search score and a semantic ranking score and caption if you use semantic ranker. We used a [select statement](#) to return just the HotelName, Description, and Tags fields.

```
{  
    "@odata.count": 18,  
    "@search.answers": [],  
    "value": [  
        {  
            "@search.score": 2.2896252,  
            "@search.rerankerScore": 2.506816864013672,  
            "@search.captions": [  
                {  
                    "text": "Head Wind Resort. Suite. coffee in lobby\r\nfree  
wifi\r\nview. The best of old town hospitality combined with views of the  
river and cool breezes off the prairie. Our penthouse suites offer views for  
miles and the rooftop plaza is open to all guests from sunset to 10 p.m.  
Enjoy a **complimentary continental breakfast** in the lobby, and free Wi-Fi  
throughout the hotel..",  
                    "highlights": ""  
                }  
            ],  
            "HotelName": "Head Wind Resort",  
            "Description": "The best of old town hospitality combined with views of the  
river and cool breezes off the prairie. Our penthouse suites offer views for  
miles and the rooftop plaza is open to all guests from sunset to 10 p.m.  
Enjoy a complimentary continental breakfast in the lobby, and free Wi-Fi  
throughout the hotel.",  
            "Tags": [  
                "coffee in lobby",  
                "free wifi",  
                "view"  
            ]  
        },  
        {  
            "@search.score": 2.2158256,  
            "@search.rerankerScore": 2.288334846496582,  
            "@search.captions": [  
                {  
                    "text": "Swan Bird Lake Inn. Budget. continental breakfast\r\nfree  
wifi\r\n24-hour front desk service. We serve a continental-style breakfast  
each morning, featuring a variety of food and drinks. Our locally made, oh-  
so-soft, caramel cinnamon rolls are a favorite with our guests. Other  
breakfast items include coffee, orange juice, milk, cereal, instant oatmeal,  
bagels, and muffins..",  
                    "highlights": ""  
                }  
            ]  
        }  
    ]  
}
```

```

        },
        ],
        "HotelName": "Swan Bird Lake Inn",
        "Description": "We serve a continental-style breakfast each morning, featuring a variety of food and drinks. Our locally made, oh-so-soft, caramel cinnamon rolls are a favorite with our guests. Other breakfast items include coffee, orange juice, milk, cereal, instant oatmeal, bagels, and muffins.",
        "Tags": [
            "continental breakfast",
            "free wifi",
            "24-hour front desk service"
        ]
    },
    {
        "@search.score": 0.92481667,
        "@search.rerankerScore": 2.221315860748291,
        "@search.captions": [
            {
                "text": "White Mountain Lodge & Suites. Resort and Spa. continental breakfast\r\npool\r\nrestaurant. Live amongst the trees in the heart of the forest. Hike along our extensive trail system. Visit the Natural Hot Springs, or enjoy our signature hot stone massage in the Cathedral of Firs. Relax in the meditation gardens, or join new friends around the communal firepit. Weekend evening entertainment on the patio features special guest musicians or poetry readings..",
                "highlights": ""
            }
        ],
        "HotelName": "White Mountain Lodge & Suites",
        "Description": "Live amongst the trees in the heart of the forest. Hike along our extensive trail system. Visit the Natural Hot Springs, or enjoy our signature hot stone massage in the Cathedral of Firs. Relax in the meditation gardens, or join new friends around the communal firepit. Weekend evening entertainment on the patio features special guest musicians or poetry readings.",
        "Tags": [
            "continental breakfast",
            "pool",
            "restaurant"
        ]
    },
    . . .
}]
}

```

Get service endpoints

In the remaining sections, you set up API calls to Azure OpenAI and Azure AI Search. Get the service endpoints so that you can provide them as variables in your code.

1. Sign in to the [Azure portal](#).

2. [Find your search service](#).
3. On the **Overview** home page, copy the URL. An example endpoint might look like `https://example.search.windows.net.`
4. [Find your Azure OpenAI service](#).
5. On the **Overview** home page, select the link to view the endpoints. Copy the URL. An example endpoint might look like `https://example.openai.azure.com/`.

Sign in to Azure

You're using Microsoft Entra ID and role assignments for the connection. Make sure you're logged in to the same tenant and subscription as Azure AI Search and Azure OpenAI. You can use the Azure CLI on the command line to show current properties, change properties, and to sign in. For more information, see [Connect without keys](#).

Run each of the following commands in sequence.

```
azure-cli  
  
az account show  
  
az account set --subscription <PUT YOUR SUBSCRIPTION ID HERE>  
  
az login --tenant <PUT YOUR TENANT ID HERE>
```

You should now be logged in to Azure from your local device.

Set up the .NET app

To follow along with the steps ahead, you can either clone the completed sample app from GitHub, or create the app yourself.

Clone the sample app

To access the completed sample app for this article:

1. Clone the [azure-search-dotnet-samples](#) repo from GitHub.

```
Bash
```

```
git clone https://github.com/Azure-Samples/azure-search-dotnet-samples
```

2. Navigate into the `quickstart-rag` folder.
3. Open the `quickstart-rag` folder in Visual Studio Code or open the solution file using Visual Studio.

Create the sample app

Complete the following steps to create a .NET console app to connect to an AI model.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

```
.NET CLI  
dotnet new console -o AISearchRag
```

2. Change directory into the app folder:

```
.NET CLI  
cd AISearchRag
```

3. Install the required packages:

```
Bash  
dotnet add package Azure.AI.OpenAI  
dotnet add package Azure.Identity  
dotnet add package Azure.Search.Documents
```

4. Open the app in Visual Studio Code (or your editor of choice).

```
Bash  
code .
```

Set up the query and chat thread

The following example demonstrates how to set up a minimal RAG scenario using Azure AI Search to provide an OpenAI model with contextual resources to improve the generated responses.

1. In the `minimal-query` project of the sample repo, open the `Program.cs` file to view the first example. If you created the project yourself, add the following code to connect to and query the Azure AI Search and Azure OpenAI services.

 Note

Make sure to replace the placeholders for the Azure OpenAI endpoint and model name, as well as the Azure AI Search endpoint and index name.

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Azure.AI.OpenAI;
using OpenAI.Chat;
using System.Text.Json;
using Microsoft.Extensions.Logging;

// Azure resource endpoints and deployment info
string azureSearchServiceEndpoint = "azure-ai-search-endpoint";
string azureOpenAIEndpoint = "azure-ai-openai-endpoint";
string azureDeploymentModel = "azure-ai-deployment-name";
string indexName = "hotels-sample-index";

// Set up Azure credentials and clients
var credential = new DefaultAzureCredential();
var searchClient = new SearchClient(new Uri(azureSearchServiceEndpoint),
indexName, credential);
var openAIClient = new AzureOpenAIClient(new Uri(azureOpenAIEndpoint),
credential);

// Prompt template for grounding the LLM response in search results
string GROUNDED_PROMPT = @"You are a friendly assistant that recommends
hotels based on activities and amenities.

Answer the query using only the sources provided below in a friendly and
concise bulleted manner

Answer ONLY with the facts listed in the list of sources below.
If there isn't enough information below, say you don't know.
Do not generate answers that don't use the sources below.

Query: {0}
Sources: {1}";

// The user's query
string query = "Can you recommend a few hotels with complimentary
breakfast?";

// Configure search options: top 5 results, select relevant fields
```

```

var options = new SearchOptions { Size = 5 };
options.Select.Add("Description");
options.Select.Add("HotelName");
options.Select.Add("Tags");

// Execute the search
var searchResults = await searchClient.SearchAsync<SearchDocument>(query,
options);
var sources = new List<string>();

await foreach (var result in searchResults.Value.GetResultsAsync())
{
    var doc = result.Document;
    // Format each result as: HotelName:Description:Tags
    sources.Add($"{doc["HotelName"]}:{doc["Description"]}:{doc["Tags"]}");
}
string sourcesFormatted = string.Join("\n", sources);

// Format the prompt with the query and sources
string formattedPrompt = string.Format(GROUNDED_PROMPT, query,
sourcesFormatted);

// Create a chat client for the specified deployment/model
ChatClient chatClient = openAIClient.GetChatClient(azureDeploymentModel);

// Send the prompt to the LLM and stream the response
var chatUpdates = chatClient.CompleteChatStreamingAsync(
    [ new UserChatMessage(formattedPrompt) ]
);

// Print the streaming response to the console
await foreach (var chatUpdate in chatUpdates)
{
    if (chatUpdate.Role.HasValue)
    {
        Console.Write($"{chatUpdate.Role} : ");
    }
    foreach (var contentPart in chatUpdate.ContentUpdate)
    {
        Console.Write(contentPart.Text);
    }
}

```

The preceding code accomplishes the following:

- Searches an Azure Search index for hotels matching a user query about complimentary breakfast, retrieving hotel name, description, and tags.
- Formats the search results into a structured list to serve as contextual sources for the generative AI model.
- Constructs a prompt instructing the Azure OpenAI model to answer using only the provided sources.

- Sends the prompt to the AI model and streams the generated response.
- Outputs the AI's response to the console, displaying both the role and content as it streams.

2. Run the project to initiate a basic RAG scenario. The output from Azure OpenAI consists of recommendations for several hotels, such as the following example:

```
Output

Sure! Here are a few hotels that offer complimentary breakfast:

- **Head Wind Resort**
- Complimentary continental breakfast in the lobby
- Free Wi-Fi throughout the hotel

- **Double Sanctuary Resort**
- Continental breakfast included

- **White Mountain Lodge & Suites**
- Continental breakfast available

- **Swan Bird Lake Inn**
- Continental-style breakfast each morning with a variety of food and drinks
such as caramel cinnamon rolls, coffee, orange juice, milk, cereal,
instant oatmeal, bagels, and muffins
```

To experiment further, change the query and rerun the last step to better understand how the model works with the grounding data. You can also modify the prompt to change the tone or structure of the output.

Troubleshooting

You might receive any of the following errors while testing:

- **Forbidden:** Check Azure AI Search configuration to make sure role-based access is enabled.
- **Authorization failed:** Wait a few minutes and try again. It can take several minutes for role assignments to become operational.
- **Resource not found:** Check the resource URIs and make sure the API version on the chat model is valid.

Send a complex RAG query

Azure AI Search supports [complex types](#) for nested JSON structures. In the hotels-sample-index, `Address` is an example of a complex type, consisting of `Address.StreetAddress`,

`Address.City`, `Address.StateProvince`, `Address.PostalCode`, and `Address.Country`. The index also has complex collection of `Rooms` for each hotel. If your index has complex types, your query can provide those fields if you first convert the search results output to JSON, and then pass the JSON to the chat model.

1. In the `complex-query` project of the sample repo, open the `Program.cs` file. If you created the project yourself, replace your code with the following:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Azure.AI.OpenAI;
using OpenAI.Chat;
using System.Text.Json;
using Microsoft.Extensions.Logging;

// Azure resource endpoints and deployment info
string azureSearchServiceEndpoint = "azure-ai-search-endpoint";
string azureOpenAIEndpoint = "azure-ai-openai-endpoint";
string azureDeploymentModel = "azure-ai-deployment-name";
string indexName = "hotels-sample-index";

// Set up Azure credentials and clients
var credential = new DefaultAzureCredential();
var searchClient = new SearchClient(new Uri(azureSearchServiceEndpoint),
indexName, credential);
var openAIClient = new AzureOpenAIClient(new Uri(azureOpenAIEndpoint),
credential);

// Prompt template for the OpenAI model
string groundedPrompt =
    @"You are a friendly assistant that recommends hotels based on activities
and amenities.

    Answer the query using only the sources provided below in a friendly and
concise bulleted manner.

    Answer ONLY with the facts listed in the list of sources below.
    If there isn't enough information below, say you don't know.
    Do not generate answers that don't use the sources below.

Query: {0}
Sources: {1}";

// The user query and fields to select from search
var query = "Can you recommend a few hotels that offer complimentary
breakfast? Tell me their description, address, tags, and the rate for one
room that sleeps 4 people.";
var selectedFields = new[] { "HotelName", "Description", "Address", "Rooms",
"Tags" };
```

```

// Configure search options
var options = new SearchOptions { Size = 5 };
foreach (var field in selectedFields)
{
    options.Select.Add(field);
}

// Run Azure Cognitive Search
var searchResults = await searchClient.SearchAsync<SearchDocument>(query,
options);

// Filter and format search results
var sourcesFiltered = new List<Dictionary<string, object>>();
await foreach (var result in searchResults.Value.GetResultsAsync())
{
    sourcesFiltered.Add(
        selectedFields
            .Where(f => result.Document.TryGetValue(f, out _))
            .ToDictionary(f => f, f => result.Document[f])
    );
}
var sourcesFormatted = string.Join("\n", sourcesFiltered.ConvertAll(source =>
JsonSerializer.Serialize(source)));

// Format the prompt for OpenAI
string formattedPrompt = string.Format(groundedPrompt, query,
sourcesFormatted);

// Get a chat client for the OpenAI deployment
ChatClient chatClient = openAIClient.GetChatClient(azureDeploymentModel);

// Send the prompt to Azure OpenAI and stream the response
var chatUpdates = chatClient.CompleteChatStreamingAsync(
    new[] { new UserChatMessage(formattedPrompt) }
);

// Output the streamed chat response
await foreach (var chatUpdate in chatUpdates)
{
    if (chatUpdate.Role.HasValue)
    {
        Console.Write($"{chatUpdate.Role} : ");
    }
    foreach (var contentPart in chatUpdate.ContentUpdate)
    {
        Console.Write(contentPart.Text);
    }
}

```

2. Run the project to initiate a basic RAG scenario. The output from Azure OpenAI consists of recommendations for several hotels, such as the following example:

Output

1. **Double Sanctuary Resort**
 - **Description**: 5-star luxury hotel with the biggest rooms in the city. Recognized as the #1 hotel in the area by Traveler magazine. Features include free WiFi, flexible check-in/out, a fitness center, and in-room espresso.
 - **Address**: 2211 Elliott Ave, Seattle, WA, 98121, USA
 - **Tags**: view, pool, restaurant, bar, continental breakfast
 - **Room Rate for 4 People**:
 - Suite, 2 Queen Beds: \$254.99 per night
2. **Starlight Suites**
 - **Description**: Spacious all-suite hotel with complimentary airport shuttle and WiFi. Facilities include an indoor/outdoor pool, fitness center, and Florida Green certification. Complimentary coffee and HDTV are also available.
 - **Address**: 19575 Biscayne Blvd, Aventura, FL, 33180, USA
 - **Tags**: pool, coffee in lobby, free wifi
 - **Room Rate for 4 People**:
 - Suite, 2 Queen Beds (Cityside): \$231.99 per night
 - Deluxe Room, 2 Queen Beds (Waterfront View): \$148.99 per night
3. **Good Business Hotel**
 - **Description**: Located one mile from the airport with free WiFi, an outdoor pool, and a complimentary airport shuttle. Close proximity to Lake Lanier and downtown. The business center includes printers, a copy machine, fax, and a work area.
 - **Address**: 4400 Ashford Dunwoody Rd NE, Atlanta, GA, 30346, USA
 - **Tags**: pool, continental breakfast, free parking
 - **Room Rate for 4 People**:
 - Budget Room, 2 Queen Beds (Amenities): \$60.99 per night
 - Deluxe Room, 2 Queen Beds (Amenities): \$139.99 per night

Troubleshooting

If you see output messages while debugging related to `ManagedIdentityCredential` and token acquisition failures, it could be that you have multiple tenants, and your Azure sign-in is using a tenant that doesn't have your search service. To get your tenant ID, search the Azure portal for "tenant properties" or run `az login tenant list`.

Once you have your tenant ID, run `az login --tenant <YOUR-TENANT-ID>` at a command prompt, and then rerun the script.

You can also log errors in your code by creating an instance of `ILogger`:

C#

```
using var loggerFactory = LoggerFactory.Create(builder =>
{
```

```
builder.AddConsole();
});
ILogger logger = loggerFactory.CreateLogger<Program>();
```

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal by using the **All resources** or **Resource groups** link in the leftmost pane.

Related content

- [Tutorial: Build a RAG solution in Azure AI Search](#)

Quickstart: Full-text search

In this quickstart, you use the `Azure.Search.Documents` client library to create, load, and query a search index with sample data for [full-text search](#). Full-text search uses Apache Lucene for indexing and queries and the BM25 ranking algorithm for scoring results.

This quickstart uses fictional hotel data from the [azure-search-sample-data](#) repo to populate the index.

💡 Tip

You can download the [source code](#) to start with a finished project or follow these steps to create your own.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) if you don't have one. For this quickstart, you can use a free service.

Microsoft Entra ID prerequisites

For the recommended keyless authentication with Microsoft Entra ID, you must:

- Install the [Azure CLI](#).
- Assign the `Search Service Contributor` and `Search Index Data Contributor` roles to your user account. You can assign roles in the Azure portal under **Access control (IAM)** > **Add role assignment**. For more information, see [Connect to Azure AI Search using roles](#).

Get service information

You need to retrieve the following information to authenticate your application with your Azure AI Search service:

Microsoft Entra ID

[+] [Expand table](#)

Variable name	Value
SEARCH_API_ENDPOINT	This value can be found in the Azure portal. Select your search service and then from the left menu, select Overview . The Url value under Essentials is the endpoint that you need. An example endpoint might look like <code>https://mydemo.search.windows.net.</code>

Learn more about [keyless authentication](#) and [setting environment variables](#).

Set up

1. Create a new folder `full-text-quickstart` to contain the application and open Visual Studio Code in that folder with the following command:

```
shell
mkdir full-text-quickstart && cd full-text-quickstart
```

2. Create a new console application with the following command:

```
shell
dotnet new console
```

3. Install the Azure AI Search client library ([Azure.Search.Documents](#)) for .NET with:

```
Console
dotnet add package Azure.Search.Documents
```

4. For the **recommended** keyless authentication with Microsoft Entra ID, install the [Azure.Identity](#) package with:

```
Console
dotnet add package Azure.Identity
```

5. For the **recommended** keyless authentication with Microsoft Entra ID, sign in to Azure with the following command:

```
Console
az login
```

Create, load, and query a search index

In the prior [set up](#) section, you created a new console application and installed the Azure AI Search client library.

In this section, you add code to create a search index, load it with documents, and run queries. You run the program to see the results in the console. For a detailed explanation of the code, see the [Explaining the code](#) section.

The sample code in this quickstart uses Microsoft Entra ID for the recommended keyless authentication. If you prefer to use an API key, you can replace the `DefaultAzureCredential` object with a `AzureKeyCredential` object.

Microsoft Entra ID

```
C#  
Uri serviceEndpoint = new Uri($"https://<Put your search service NAME here>.search.windows.net/");  
DefaultAzureCredential credential = new();
```

1. In `Program.cs`, paste the following code. Edit the `serviceName` and `apiKey` variables with your search service name and admin API key.

```
C#  
  
using System;  
using Azure;  
using Azure.Identity;  
using Azure.Search.Documents;  
using Azure.Search.Documents.Indexes;  
using Azure.Search.Documents.Indexes.Models;  
using Azure.Search.Documents.Models;  
  
namespace AzureSearch.Quickstart  
  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Your search service endpoint  
            Uri serviceEndpoint = new Uri($"https://<Put your search service NAME here>.search.windows.net/");  
  
            // Use the recommended keyless credential instead of the
```

```
AzureKeyCredential credential.  
    DefaultAzureCredential credential = new();  
    //AzureKeyCredential credential = new AzureKeyCredential("Your  
    search service admin key");  
  
    // Create a SearchIndexClient to send create/delete index commands  
    SearchIndexClient searchIndexClient = new  
    SearchIndexClient(serviceEndpoint, credential);  
  
    // Create a SearchClient to load and query documents  
    string indexName = "hotels-quickstart";  
    SearchClient searchClient = new SearchClient(serviceEndpoint,  
indexName, credential);  
  
    // Delete index if it exists  
    Console.WriteLine("{0}", "Deleting index...\\n");  
    DeleteIndexIfExists(indexName, searchIndexClient);  
  
    // Create index  
    Console.WriteLine("{0}", "Creating index...\\n");  
    CreateIndex(indexName, searchIndexClient);  
  
    SearchClient ingestClient =  
searchIndexClient.GetSearchClient(indexName);  
  
    // Load documents  
    Console.WriteLine("{0}", "Uploading documents...\\n");  
    UploadDocuments(ingestClient);  
  
    // Wait 2 seconds for indexing to complete before starting queries  
(for demo and console-app purposes only)  
    Console.WriteLine("Waiting for indexing...\\n");  
    System.Threading.Thread.Sleep(2000);  
  
    // Call the RunQueries method to invoke a series of queries  
    Console.WriteLine("Starting queries...\\n");  
    RunQueries(searchClient);  
  
    // End the program  
    Console.WriteLine("{0}", "Complete. Press any key to end this  
program...\\n");  
    Console.ReadKey();  
}  
  
// Delete the hotels-quickstart index to reuse its name  
private static void DeleteIndexIfExists(string indexName,  
SearchIndexClient searchIndexClient)  
{  
    searchIndexClient.GetIndexNames();  
    {  
        searchIndexClient.DeleteIndex(indexName);  
    }  
}  
// Create hotels-quickstart index  
private static void CreateIndex(string indexName, SearchIndexClient
```

```

searchIndexClient)

{
    FieldBuilder fieldBuilder = new FieldBuilder();
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var definition = new SearchIndex(indexName, searchFields);

    var suggester = new SearchSuggester("sg", new[] { "HotelName",
"Category", "Address/City", "Address/StateProvince" });
    definition.Suggesters.Add(suggester);

    searchIndexClient.CreateIndex(definition);
}

// Upload documents in a single Upload request.
private static void UploadDocuments(SearchClient searchClient)
{
    IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                HotelName = "Stay-Kay City Hotel",
                Description = "This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.",
                Category = "Boutique",
                Tags = new[] { "view", "air conditioning", "concierge" },
                ParkingIncluded = false,
                LastRenovationDate = new DateTimeOffset(2022, 1, 18, 0, 0, 0, TimeSpan.Zero),
                Rating = 3.6,
                Address = new Address()
                {
                    StreetAddress = "677 5th Ave",
                    City = "New York",
                    StateProvince = "NY",
                    PostalCode = "10022",
                    Country = "USA"
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "2",
                HotelName = "Old Century Hotel",
                Description = "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music."
            })
    );
}

```

```

Category = "Boutique",
Tags = new[] { "pool", "free wifi", "concierge" },
ParkingIncluded = false,
LastRenovationDate = new DateTimeOffset(2019, 2, 18, 0,
0, 0, TimeSpan.Zero),
Rating = 3.60,
Address = new Address()
{
    StreetAddress = "140 University Town Center Dr",
    City = "Sarasota",
    StateProvince = "FL",
    PostalCode = "34243",
    Country = "USA"
}
}),
IndexDocumentsAction.Upload(
new Hotel()
{
    HotelId = "3",
    HotelName = "Gastronomic Landscape Hotel",
    Description = "The Gastronomic Hotel stands out for its
culinary excellence under the management of William Dough, who advises on and
oversees all of the Hotel's restaurant services.",
    Category = "Suite",
    Tags = new[] { "restaurant", "bar", "continental
breakfast" },
    ParkingIncluded = true,
    LastRenovationDate = new DateTimeOffset(2015, 9, 20, 0,
0, 0, TimeSpan.Zero),
    Rating = 4.80,
    Address = new Address()
{
    StreetAddress = "3393 Peachtree Rd",
    City = "Atlanta",
    StateProvince = "GA",
    PostalCode = "30326",
    Country = "USA"
}
}),
IndexDocumentsAction.Upload(
new Hotel()
{
    HotelId = "4",
    HotelName = "Sublime Palace Hotel",
    Description = "Sublime Palace Hotel is located in the
heart of the historic center of Sublime in an extremely vibrant and lively area
within short walking distance to the sites and landmarks of the city and is
surrounded by the extraordinary beauty of churches, buildings, shops and
monuments. Sublime Cliff is part of a lovingly restored 19th century resort,
updated for every modern convenience.",
    Category = "Boutique",
    Tags = new[] { "concierge", "view", "air conditioning"
},
    ParkingIncluded = true,
    LastRenovationDate = new DateTimeOffset(2020, 2, 06, 0,
0, 0, TimeSpan.Zero)
}
)
);

```

```
0, 0, TimeSpan.Zero),
    Rating = 4.60,
    Address = new Address()
{
    StreetAddress = "7400 San Pedro Ave",
    City = "San Antonio",
    StateProvince = "TX",
    PostalCode = "78216",
    Country = "USA"
}
})
);

try
{
    IndexDocumentsResult result =
searchClient.IndexDocuments(batch);
}
catch (Exception)
{
    // If for some reason any documents are dropped during
indexing, you can compensate by delaying and
    // retrying. This simple demo just logs the failed document
keys and continues.
    Console.WriteLine("Failed to index some of the documents:
{0}");
}
}

// Run queries, use WriteDocuments to print output
private static void RunQueries(SearchClient searchClient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    // Query 1
    Console.WriteLine("Query #1: Search on empty term '*' to return all
documents, showing a subset of fields...\n");

    options = new SearchOptions()
{
    IncludeTotalCount = true,
    Filter = "",
    OrderBy = { "" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Rating");

response = searchClient.Search<Hotel>("*", options);
WriteDocuments(response);

// Query 2
Console.WriteLine("Query #2: Search on 'hotels', filter on 'Rating")
```

```

gt 4', sort by Rating in descending order...\n");

options = new SearchOptions()
{
    Filter = "Rating gt 4",
    OrderBy = { "Rating desc" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Rating");

response = searchClient.Search<Hotel>("hotels", options);
WriteDocuments(response);

// Query 3
Console.WriteLine("Query #3: Limit search to specific fields (pool
in Tags field)...\\n");

options = new SearchOptions()
{
    SearchFields = { "Tags" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Tags");

response = searchClient.Search<Hotel>("pool", options);
WriteDocuments(response);

// Query 4 - Use Facets to return a faceted navigation structure
for a given query
    // Filters are typically used with facets to narrow results on
OnClick events
    Console.WriteLine("Query #4: Facet on 'Category'...\\n");

options = new SearchOptions()
{
    Filter = ""
};

options.Facets.Add("Category");

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Category");

response = searchClient.Search<Hotel>("*", options);
WriteDocuments(response);

// Query 5
Console.WriteLine("Query #5: Look up a specific document...\\n");

Response<Hotel> lookupResponse;

```

```

        lookupResponse = searchClient.GetDocument<Hotel>("3");

        Console.WriteLine(lookupResponse.Value.HotelId);

        // Query 6
        Console.WriteLine("Query #6: Call Autocomplete on HotelName...\n");

        var autoresponse = searchClient.Autocomplete("sa", "sg");
        WriteDocuments(autoresponse);

    }

    // Write search results to console
    private static void WriteDocuments(SearchResults<Hotel> searchResults)
    {
        foreach (SearchResult<Hotel> result in searchResults.GetResults())
        {
            Console.WriteLine(result.Document);
        }

        Console.WriteLine();
    }

    private static void WriteDocuments(AutocompleteResults autoResults)
    {
        foreach (AutocompleteItem result in autoResults.Results)
        {
            Console.WriteLine(result.Text);
        }

        Console.WriteLine();
    }
}
}

```

2. In the same folder, create a new file named *Hotel.cs* and paste the following code. This code defines the structure of a hotel document.

C#

```

using System;
using System.Text.Json.Serialization;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;

namespace AzureSearch.Quickstart
{
    public partial class Hotel
    {
        [SimpleField(IsKey = true, IsFilterable = true)]
        public string HotelId { get; set; }
    }
}

```

```

[SearchableField(IsSortable = true)]
public string HotelName { get; set; }

[SearchableField(AnalyzerName = LexicalAnalyzerName.Values.EnLucene)]
public string Description { get; set; }

[SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
public string Category { get; set; }

[SearchableField(IsFilterable = true, IsFacetable = true)]
public string[] Tags { get; set; }

[SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
public bool? ParkingIncluded { get; set; }

[SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
public DateTimeOffset? LastRenovationDate { get; set; }

[SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
public double? Rating { get; set; }

[SearchableField]
public Address Address { get; set; }
}
}

```

3. Create a new file named *Hotel.cs* and paste the following code to define the structure of a hotel document. Attributes on the field determine how it's used in an application. For example, the `IsFilterable` attribute must be assigned to every field that supports a filter expression.

C#

```

using System;
using System.Text.Json.Serialization;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;

namespace AzureSearch.Quickstart
{
    public partial class Hotel
    {
        [SimpleField(IsKey = true, IsFilterable = true)]
        public string HotelId { get; set; }

        [SearchableField(IsSortable = true)]
        public string HotelName { get; set; }
    }
}

```

```

        [SearchableField(AnalyzerName = LexicalAnalyzerName.Values.EnLucene)]
        public string Description { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public string Category { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string[] Tags { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public bool? ParkingIncluded { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public DateTimeOffset? LastRenovationDate { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public double? Rating { get; set; }

        [SearchableField]
        public Address Address { get; set; }
    }
}

```

4. Create a new file named `Address.cs` and paste the following code to define the structure of an address document.

C#

```

using Azure.Search.Documents.Indexes;

namespace AzureSearch.Quickstart
{
    public partial class Address
    {
        [SearchableField(IsFilterable = true)]
        public string StreetAddress { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public string City { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public string StateProvince { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
        public string PostalCode { get; set; }
    }
}

```

```
[SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
    public string Country { get; set; }
}
```

5. Create a new file named *Hotel.Methods.cs* and paste the following code to define a `ToString()` override for the `Hotel` class.

C#

```
using System;
using System.Text;

namespace AzureSearch.Quickstart
{
    public partial class Hotel
    {
        public override string ToString()
        {
            var builder = new StringBuilder();

            if (!String.IsNullOrEmpty(HotelId))
            {
                builder.AppendFormat("HotelId: {0}\n", HotelId);
            }

            if (!String.IsNullOrEmpty(HotelName))
            {
                builder.AppendFormat("Name: {0}\n", HotelName);
            }

            if (!String.IsNullOrEmpty(Description))
            {
                builder.AppendFormat("Description: {0}\n", Description);
            }

            if (!String.IsNullOrEmpty(Category))
            {
                builder.AppendFormat("Category: {0}\n", Category);
            }

            if (Tags != null && Tags.Length > 0)
            {
                builder.AppendFormat("Tags: [ {0} ]\n", String.Join(", ",
Tags));
            }

            if (ParkingIncluded.HasValue)
            {
                builder.AppendFormat("Parking included: {0}\n",
ParkingIncluded.Value ? "yes" : "no");
            }
        }
    }
}
```

```

        if (LastRenovationDate.HasValue)
        {
            builder.AppendFormat("Last renovated on: {0}\n",
LastRenovationDate);
        }

        if (Rating.HasValue)
        {
            builder.AppendFormat("Rating: {0}\n", Rating);
        }

        if (Address != null && !Address.IsEmpty)
        {
            builder.AppendFormat("Address: \n{0}\n", Address.ToString());
        }

        return builder.ToString();
    }
}
}

```

6. Create a new file named `Address.Methods.cs` and paste the following code to define a `ToString()` override for the `Address` class.

C#

```

using System;
using System.Text;
using System.Text.Json.Serialization;

namespace AzureSearch.Quickstart
{
    public partial class Address
    {
        public override string ToString()
        {
            var builder = new StringBuilder();

            if (!IsEmpty)
            {
                builder.AppendFormat("{0}\n{1}, {2} {3}\n{4}", StreetAddress,
City, StateProvince, PostalCode, Country);
            }

            return builder.ToString();
        }

        [JsonIgnore]
        public bool IsEmpty => String.IsNullOrEmpty(StreetAddress) &&
                           String.IsNullOrEmpty(City) &&
                           String.IsNullOrEmpty(StateProvince) &&
                           String.IsNullOrEmpty(PostalCode) &&

```

```
        String.IsNullOrEmpty(Country));
    }
}
```

7. Build and run the application with the following command:

```
shell  
dotnet run
```

Output includes messages from `Console.WriteLine`, with the addition of query information and results.

Explaining the code

In the previous sections, you created a new console application and installed the Azure AI Search client library. You added code to create a search index, load it with documents, and run queries. You ran the program to see the results in the console.

In this section, we explain the code you added to the console application.

Create a search client

In `Program.cs`, you created two clients:

- `SearchIndexClient` creates the index.
- `SearchClient` loads and queries an existing index.

Both clients need the search service endpoint and credentials described previously in the [Get service information](#) section.

The sample code in this quickstart uses Microsoft Entra ID for the recommended keyless authentication. If you prefer to use an API key, you can replace the `DefaultAzureCredential` object with a `AzureKeyCredential` object.

Microsoft Entra ID

C#

```
Uri serviceEndpoint = new Uri($"https://<Put your search service NAME  
here>.search.windows.net/");  
DefaultAzureCredential credential = new();
```

C#

```
static void Main(string[] args)
{
    // Your search service endpoint
    Uri serviceEndpoint = new Uri($"https://<Put your search service NAME here>.search.windows.net/");

    // Use the recommended keyless credential instead of the AzureKeyCredential
    // credential.
    DefaultAzureCredential credential = new();
    //AzureKeyCredential credential = new AzureKeyCredential("Your search service
    admin key");

    // Create a SearchIndexClient to send create/delete index commands
    SearchIndexClient searchIndexClient = new SearchIndexClient(serviceEndpoint,
    credential);

    // Create a SearchClient to load and query documents
    string indexName = "hotels-quickstart";
    SearchClient searchClient = new SearchClient(serviceEndpoint, indexName,
    credential);

    // REDACTED FOR BREVITY . . .
}
```

Create an index

This quickstart builds a Hotels index that you load with hotel data and execute queries against. In this step, you define the fields in the index. Each field definition includes a name, data type, and attributes that determine how the field is used.

In this example, synchronous methods of the *Azure.Search.Documents* library are used for simplicity and readability. However, for production scenarios, you should use asynchronous methods to keep your app scalable and responsive. For example, you would use [CreateIndexAsync](#) instead of [CreateIndex](#).

Define the structures

You created two helper classes, *Hotel.cs* and *Address.cs*, to define the structure of a hotel document and its address. The `Hotel` class includes fields for a hotel ID, name, description, category, tags, parking, renovation date, rating, and address. The `Address` class includes fields for street address, city, state/province, postal code, and country/region.

In the *Azure.Search.Documents* client library, you can use [SearchableField](#) and [SimpleField](#) to streamline field definitions. Both are derivatives of a [SearchField](#) and can potentially simplify your code:

- `SimpleField` can be any data type, is always non-searchable (ignored for full text search queries), and is retrievable (not hidden). Other attributes are off by default, but can be enabled. You might use a `SimpleField` for document IDs or fields used only in filters, facets, or scoring profiles. If so, be sure to apply any attributes that are necessary for the scenario, such as `IsKey = true` for a document ID. For more information, see [SimpleFieldAttribute.cs](#) in source code.
- `SearchableField` must be a string, and is always searchable and retrievable. Other attributes are off by default, but can be enabled. Because this field type is searchable, it supports synonyms and the full complement of analyzer properties. For more information, see the [SearchableFieldAttribute.cs](#) in source code.

Whether you use the basic `SearchField` API or either one of the helper models, you must explicitly enable filter, facet, and sort attributes. For example, `IsFilterable`, `IsSortable`, and `IsFacetable` must be explicitly attributed, as shown in the previous sample.

Create the search index

In `Program.cs`, you create a `SearchIndex` object, and then call the `CreateIndex` method to express the index in your search service. The index also includes a `SearchSuggester` to enable autocomplete on the specified fields.

C#

```
// Create hotels-quickstart index
private static void CreateIndex(string indexName, SearchIndexClient
searchIndexClient)
{
    FieldBuilder fieldBuilder = new FieldBuilder();
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var definition = new SearchIndex(indexName, searchFields);

    var suggester = new SearchSuggester("sg", new[] { "HotelName", "Category",
"Address/City", "Address/StateProvince" });
    definition.Suggesters.Add(suggester);

    searchIndexClient.CreateOrUpdateIndex(definition);
}
```

Load documents

Azure AI Search searches over content stored in the service. In this step, you load JSON documents that conform to the hotel index you created.

In Azure AI Search, search documents are data structures that are both inputs to indexing and outputs from queries. As obtained from an external data source, document inputs might be rows in a database, blobs in Blob storage, or JSON documents on disk. In this example, we're taking a shortcut and embedding JSON documents for four hotels in the code itself.

When uploading documents, you must use an [IndexDocumentsBatch](#) object. An [IndexDocumentsBatch](#) object contains a collection of [Actions](#), each of which contains a document and a property telling Azure AI Search what action to perform ([upload](#), [merge](#), [delete](#), and [mergeOrUpload](#)).

In *Program.cs*, you create an array of documents and index actions, and then pass the array to [IndexDocumentsBatch](#). The following documents conform to the `hotels-quickstart` index, as defined by the `Hotel` class.

C#

```
// Upload documents in a single Upload request.
private static void UploadDocuments(SearchClient searchClient)
{
    IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                HotelName = "Stay-Kay City Hotel",
                Description = "This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.",
                Category = "Boutique",
                Tags = new[] { "view", "air conditioning", "concierge" },
                ParkingIncluded = false,
                LastRenovationDate = new DateTimeOffset(2022, 1, 18, 0, 0, 0,
TimeSpan.Zero),
                Rating = 3.6,
                Address = new Address()
                {
                    StreetAddress = "677 5th Ave",
                    City = "New York",
                    StateProvince = "NY",
                    PostalCode = "10022",
                    Country = "USA"
                }
            ),
            // REDACTED FOR BREVITY
        )
    );
}
```

Once you initialize the [IndexDocumentsBatch](#) object, you can send it to the index by calling [IndexDocuments](#) on your [SearchClient](#) object.

You load documents using `SearchClient` in `Main()`, but the operation also requires admin rights on the service, which is typically associated with `SearchIndexClient`. One way to set up this operation is to get `SearchClient` through `SearchIndexClient` (`searchIndexClient` in this example).

C#

```
SearchClient ingestorClient = searchIndexClient.GetSearchClient(indexName);

// Load documents
Console.WriteLine("{0}", "Uploading documents...\n");
UploadDocuments(ingesterClient);
```

Because we have a console app that runs all commands sequentially, we add a 2-second wait time between indexing and queries.

C#

```
// Wait 2 seconds for indexing to complete before starting queries (for demo and
// console-app purposes only)
Console.WriteLine("Waiting for indexing...\n");
System.Threading.Thread.Sleep(2000);
```

The 2-second delay compensates for indexing, which is asynchronous, so that all documents can be indexed before the queries are executed. Coding in a delay is typically only necessary in demos, tests, and sample applications.

Search an index

You can get query results as soon as the first document is indexed, but actual testing of your index should wait until all documents are indexed.

This section adds two pieces of functionality: query logic, and results. For queries, use the `Search` method. This method takes search text (the query string) and other [options](#).

The `SearchResults` class represents the results.

In `Program.cs`, the `WriteDocuments` method prints search results to the console.

C#

```
// Write search results to console
private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.GetResults())
    {
```

```

        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}

private static void WriteDocuments(AutocompleteResults autoResults)
{
    foreach (AutocompleteItem result in autoResults.Results)
    {
        Console.WriteLine(result.Text);
    }

    Console.WriteLine();
}

```

Query example 1

The `RunQueries` method executes queries and returns results. Results are Hotel objects. This sample shows the method signature and the first query. This query demonstrates the `Select` parameter that lets you compose the result using selected fields from the document.

C#

```

// Run queries, use WriteDocuments to print output
private static void RunQueries(SearchClient searchClient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    // Query 1
    Console.WriteLine("Query #1: Search on empty term '*' to return all documents,
showing a subset of fields...\n");

    options = new SearchOptions()
    {
        IncludeTotalCount = true,
        Filter = "",
        OrderBy = { "" }
    };

    options.Select.Add("HotelId");
    options.Select.Add("HotelName");
    options.Select.Add("Address/City");

    response = searchClient.Search<Hotel>("*", options);
    WriteDocuments(response);
    // REDACTED FOR BREVITY
}

```

Query example 2

In the second query, search on a term, add a filter that selects documents where *Rating* is greater than 4, and then sort by Rating in descending order. Filter is a boolean expression that is evaluated over [IsFilterable](#) fields in an index. Filter queries either include or exclude values. As such, there's no relevance score associated with a filter query.

```
C#
```

```
// Query 2
Console.WriteLine("Query #2: Search on 'hotels', filter on 'Rating gt 4', sort by
Rating in descending order...\n");

options = new SearchOptions()
{
    Filter = "Rating gt 4",
    OrderBy = { "Rating desc" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Rating");

response = searchClient.Search<Hotel>("hotels", options);
WriteDocuments(response);
```

Query example 3

The third query demonstrates `searchFields`, used to scope a full text search operation to specific fields.

```
C#
```

```
// Query 3
Console.WriteLine("Query #3: Limit search to specific fields (pool in Tags
field)...\\n");

options = new SearchOptions()
{
    SearchFields = { "Tags" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Tags");

response = searchClient.Search<Hotel>("pool", options);
WriteDocuments(response);
```

Query example 4

The fourth query demonstrates `facets`, which can be used to structure a faceted navigation structure.

```
C#
```

```
// Query 4
Console.WriteLine("Query #4: Facet on 'Category'...\n");

options = new SearchOptions()
{
    Filter = ""
};

options.Facets.Add("Category");

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Category");

response = searchClient.Search<Hotel>("*", options);
WriteDocuments(response);
```

Query example 5

In the fifth query, return a specific document. A document lookup is a typical response to `OnClick` event in a result set.

```
C#
```

```
// Query 5
Console.WriteLine("Query #5: Look up a specific document...\n");

Response<Hotel> lookupResponse;
lookupResponse = searchClient.GetDocument<Hotel>("3");

Console.WriteLine(lookupResponse.Value.HotelId);
```

Query example 6

The last query shows the syntax for autocomplete, simulating a partial user input of `sa` that resolves to two possible matches in the `sourceFields` associated with the suggester you defined in the index.

```
C#
```

```
// Query 6
Console.WriteLine("Query #6: Call Autocomplete on HotelName that starts with
'sa'...\n");

var autoreponse = searchClient.Autocomplete("sa", "sg");
WriteDocuments(autoreponse);
```

Summary of queries

The previous queries show multiple [ways of matching terms in a query](#): full-text search, filters, and autocomplete.

Full text search and filters are performed using the `SearchClient.Search` method. A search query can be passed in the `searchText` string, while a filter expression can be passed in the `Filter` property of the `SearchOptions` class. To filter without searching, just pass `"*"` for the `searchText` parameter of the `Search` method. To search without filtering, leave the `Filter` property unset, or don't pass in a `SearchOptions` instance at all.

Clean up resources

When working in your own subscription, it's a good idea to finish a project by determining whether you still need the resources you created. Resources that are left running can cost you money. You can delete resources individually, or you can delete the resource group to delete the entire set of resources.

In the [Azure portal](#), you can find and manage resources by selecting **All resources** or **Resource groups** from the left pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the Azure portal to stay under the limit.

Related content

- [Full-text search in Azure AI Search](#)
- [Examples of simple search queries](#)
- [Examples of full Lucene search syntax](#)

Quickstart: Semantic ranking

In this quickstart, you learn how to use [semantic ranking](#) by adding a semantic configuration to a search index and adding semantic parameters to a query. You can use the `hotels-sample-index` or one of your own.

In Azure AI Search, semantic ranking is query-side functionality that uses machine reading comprehension from Microsoft to rescore search results, promoting the most semantically relevant matches to the top of the list. Depending on the content and the query, semantic ranking can [significantly improve search relevance](#) with minimal developer effort.

You can add a semantic configuration to an existing index with no rebuild requirement. Semantic ranking is most effective on text that's informational or descriptive.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#) with [semantic ranker enabled](#).
- A [new or existing index](#) with descriptive or verbose text fields that are attributed as retrievable. This quickstart assumes the [hotels-sample-index](#).

Configure access

You can connect to your Azure AI Search service using API keys or Microsoft Entra ID with role assignments. Keys are easier to start with, but roles are more secure. For more information, see [Connect to Azure AI Search using roles](#).

To configure role-based access:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Settings > Keys**.
3. Under **API Access control**, select **Role-based access control** or **Both** if you need time to transition clients to role-based access.
4. From the left pane, select **Access control (IAM)**.
5. Select **Add > Add role assignment**.
6. Assign the **Search Service Contributor** and **Search Index Data Contributor** roles to your user account.

Start with an index

This quickstart assumes an existing index and modifies it to include a semantic configuration. We recommend the [hotels-sample-index](#) that you can create in minutes using an Azure portal wizard.

To start with an existing index:

1. Sign in to the [Azure portal](#) and find your search service.
2. Under **Search management > Indexes**, select the hotels-sample-index.
3. Select **Semantic configurations** to ensure the index doesn't have a semantic configuration.

hotels-sample-index ...

Save Discard Refresh Create demo app Edit JSON Delete Encryption

Documents Total storage Vector index quota usage Max storage

50 561.54 KB 0 Bytes 15 GB

Search explorer Fields CORS Scoring profiles Semantic configurations Vector profiles

Add semantic configuration Delete

You haven't created any semantic configurations

Create

4. Select **Search explorer**, and then select the **JSON view**.
5. Paste the following JSON into the query editor.

```
JSON
{
  "search": "walking distance to live music",
  "select": "HotelId, HotelName, Description",
```

```
        "count": true  
    }
```

The screenshot shows the Microsoft Azure portal interface for the 'hotels-sample-index' search service. At the top, there's a navigation bar with 'Microsoft Azure', a search bar, and a 'Copilot' button. Below the navigation, the service name 'hotels-sample-index' is displayed. A toolbar with various actions like 'Save', 'Discard', 'Refresh', 'Create demo app', 'Edit JSON', 'Delete', and 'Encryption' is visible. Key statistics are shown: 'Documents' (50), 'Total storage' (561.25 KB), 'Vector index quota usage' (0 Bytes), and 'Max storage' (15 GB). Below these, tabs for 'Search explorer', 'Fields', 'CORS', 'Scoring profiles', 'Semantic configurations', and 'Vector profiles' are present. A dropdown menu shows the date '2025-05-01-preview' and a 'View' option. The main area is a 'JSON query editor' containing the following code:

```
1 {  
2     "search": "walking distance to live music",  
3     "select": "HotelId, HotelName, Description",  
4     "count": true  
5 }
```

To the right of the editor, a 'Results' section is partially visible. A dropdown menu on the right side of the interface shows three options: 'Query view', 'Image view', and 'JSON view'. The 'JSON view' option is highlighted with a red box.

6. Select **Search** to run the query.

This query is a keyword search. The response should be similar to the following example, as scored by the default BM25 L1 ranker for full-text search.

For readability, the example only selects the `HotelId`, `HotelName`, and `Description` fields. The results contain verbatim matches on the query terms (`walking`, `distance`, `live`, `music`) or linguistic variants (`walk`, `living`).

The screenshot shows the 'JSON' results panel. It displays a single document from the search results. The document contains the following fields and their values:

```
"@odata.count": 13,  
"value": [  
    {  
        "@search.score": 5.5153193,  
        "HotelId": "2",  
        "HotelName": "Old Century Hotel",  
        "Description": "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music."  
    },  
    {
```

```
    "@search.score": 5.074317,
    "HotelId": "24",
    "HotelName": "Uptown Chic Hotel",
    "Description": "Chic hotel near the city. High-rise hotel in downtown, within walking distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum by day, and then head over to Benaroya Hall to catch the evening's concert performance."
},
{
    "@search.score": 4.8959594,
    "HotelId": "4",
    "HotelName": "Sublime Palace Hotel",
    "Description": "Sublime Cliff Hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking distance to the sites and landmarks of the city and is surrounded by the extraordinary beauty of churches, buildings, shops and monuments. Sublime Cliff is part of a lovingly restored 19th century resort, updated for every modern convenience."
},
{
    "@search.score": 2.5966604,
    "HotelId": "35",
    "HotelName": "Bellevue Suites",
    "Description": "Comfortable city living in the very center of downtown Bellevue. Newly reimagined, this hotel features apartment-style suites with sleeping, living and work spaces. Located across the street from the Light Rail to downtown. Free shuttle to the airport."
},
{
    "@search.score": 2.566386,
    "HotelId": "47",
    "HotelName": "Country Comfort Inn",
    "Description": "Situated conveniently at the north end of the village, the inn is just a short walk from the lake, offering reasonable rates and all the comforts home inlcuding living room suites and functional kitchens. Pets are welcome."
},
{
    "@search.score": 2.2405157,
    "HotelId": "9",
    "HotelName": "Smile Up Hotel",
    "Description": "Experience the fresh, modern downtown. Enjoy updated rooms, bold style & prime location. Don't miss our weekend live music series featuring who's new/next on the scene."
},
{
    "@search.score": 2.1737604,
    "HotelId": "8",
    "HotelName": "Foot Happy Suites",
    "Description": "Downtown in the heart of the business district. Close to everything. Leave your car behind and walk to the park, shopping, and restaurants. Or grab one of our bikes and take your explorations a little further."
},
{
```

```
    "@search.score": 2.0364518,
    "HotelId": "31",
    "HotelName": "Country Residence Hotel",
    "Description": "All of the suites feature full-sized kitchens stocked with cookware, separate living and sleeping areas and sofa beds. Some of the larger rooms have fireplaces and patios or balconies. Experience real country hospitality in the heart of bustling Nashville. The most vibrant music scene in the world is just outside your front door."
},
{
    "@search.score": 1.7595702,
    "HotelId": "49",
    "HotelName": "Swirling Currents Hotel",
    "Description": "Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center. Each room comes equipped with a microwave, a coffee maker and a minifridge. In-room entertainment includes complimentary W-Fi and flat-screen TVs. "
},
{
    "@search.score": 1.5502293,
    "HotelId": "15",
    "HotelName": "By the Market Hotel",
    "Description": "Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service."
},
{
    "@search.score": 1.3302404,
    "HotelId": "42",
    "HotelName": "Rock Bottom Resort & Campground",
    "Description": "Rock Bottom is nestled on 20 unspoiled acres on a private cove of Rock Bottom Lake. We feature both lodging and campground accommodations to suit just about every taste. Even though we are out of the traffic of the city, getting there is only a short drive away."
},
{
    "@search.score": 0.9050383,
    "HotelId": "38",
    "HotelName": "Lakeside B & B",
    "Description": "Nature is Home on the beach. Explore the shore by day, and then come home to our shared living space to relax around a stone fireplace, sip something warm, and explore the library by night. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackouts may apply."
},
{
    "@search.score": 0.7334347,
    "HotelId": "39",
    "HotelName": "White Mountain Lodge & Suites",
    "Description": "Live amongst the trees in the heart of the forest. Hike along our extensive trail system. Visit the Natural Hot Springs, or enjoy our signature hot stone massage in the Cathedral of Firs. Relax in the meditation gardens, or join new friends around the communal firepit. Weekend evening entertainment on the patio features special guest musicians or poetry readings."
}
```

```
    }  
]
```

This query shows how the response looks *before* semantic ranking is applied. Later, you can run the same query after semantic ranking is configured to see how the response changes.

 **Tip**

You can add a semantic configuration in the Azure portal. However, if you want to learn how to add a semantic configuration programmatically, continue with this quickstart.

Set up the client

In this quickstart, you use an IDE and the [Azure.Search.Documents](#) client library to add semantic ranking to an existing search index.

We recommend [Visual Studio](#) for this quickstart.

 **Tip**

You can download the [source code](#) to start with a finished project or follow these steps to create your own.

Install libraries

1. Start Visual Studio and open the [quickstart-semantic-search.sln](#) or create a new project using a console application template.
2. In **Tools > NuGet Package Manager**, select **Manage NuGet Packages for Solution....**
3. Select **Browse**.
4. Search for the [Azure.Search.Documents package](#) and select the latest stable version.
5. Search for the [Azure.Identity package](#) and select the latest stable version.
6. Select **Install** to add the assembly to your project and solution.

Sign in to Azure

If you signed in to the [Azure portal](#), you're signed into Azure. If you aren't sure, use the Azure CLI or Azure PowerShell to log in: `az login` or `az connect`. If you have multiple tenants

and subscriptions, see [Quickstart: Connect without keys](#) for help on how to connect.

Update the index

In this section, you update a search index to include a semantic configuration. The code gets the index definition from the search service and adds a semantic configuration.

1. Open the [BuildIndex project](#) in Visual Studio. The program consists of the following code.

This code uses a `SearchIndexClient` to update an index on your search service.

C#

```
class BuildIndex
{
    static async Task Main(string[] args)
    {
        string searchServiceName = "PUT-YOUR-SEARCH-SERVICE-NAME-HERE";
        string indexName = "hotels-sample-index";
        string endpoint = $"https://{{searchServiceName}}.search.windows.net";
        var credential = new Azure.Identity.DefaultAzureCredential();

        await ListIndexesAsync(endpoint, credential);
        await UpdateIndexAsync(endpoint, credential, indexName);
    }

    // Print a list of all indexes on the search service
    // You should see hotels-sample-index in the list
    static async Task ListIndexesAsync(string endpoint,
        Azure.Core.TokenCredential credential)
    {
        try
        {
            var indexClient = new
                Azure.Search.Documents.Indexes.SearchIndexClient(
                    new Uri(endpoint),
                    credential
                );

            var indexes = indexClient.GetIndexesAsync();

            Console.WriteLine("Here's a list of all indexes on the search
service. You should see hotels-sample-index:");
            await foreach (var index in indexes)
            {
                Console.WriteLine(index.Name);
            }
            Console.WriteLine(); // Add an empty line for readability
        }
        catch (Exception ex)
```

```

        {
            Console.WriteLine($"Error listing indexes: {ex.Message}");
        }
    }

    static async Task UpdateIndexAsync(string endpoint,
Azure.Core.TokenCredential credential, string indexName)
{
    try
    {
        var indexClient = new
Azure.Search.Documents.Indexes.SearchIndexClient(
            new Uri(endpoint),
            credential
        );

        // Get the existing definition of hotels-sample-index
        var indexResponse = await indexClient.GetIndexAsync(indexName);
        var index = indexResponse.Value;

        // Add a semantic configuration
        const string semanticConfigName = "semantic-config";
        AddSemanticConfiguration(index, semanticConfigName);

        // Update the index with the new information
        var updatedIndex = await
indexClient.CreateOrUpdateIndexAsync(index);
        Console.WriteLine("Index updated successfully.");

        // Print the updated index definition as JSON
        var refreshedIndexResponse = await
indexClient.GetIndexAsync(indexName);
        var refreshedIndex = refreshedIndexResponse.Value;
        var jsonOptions = new JsonSerializerOptions { WriteIndented = true
};

        string indexJson = JsonSerializer.Serialize(refreshedIndex,
jsonOptions);
        Console.WriteLine($"Here is the revised index
definition:\n{indexJson}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error updating index: {ex.Message}");
    }
}

// This is the semantic configuration definition
static void AddSemanticConfiguration(SearchIndex index, string
semanticConfigName)
{
    if (index.SemanticSearch == null)
    {
        index.SemanticSearch = new SemanticSearch();
    }
    var configs = index.SemanticSearch.Configurations;
}

```

```

    if (configs == null)
    {
        throw new InvalidOperationException("SemanticSearch.Configurations
is null and cannot be assigned. Your service must be Basic tier or higher.");
    }
    if (!configs.Any(c => c.Name == semanticConfigName))
    {
        var prioritizedFields = new SemanticPrioritizedFields
        {
            TitleField = new SemanticField("HotelName"),
            ContentFields = { new SemanticField("Description") },
            KeywordsFields = { new SemanticField("Tags") }
        };

        configs.Add(
            new SemanticConfiguration(
                semanticConfigName,
                prioritizedFields
            )
        );
        Console.WriteLine($"Added new semantic configuration
'{semanticConfigName}' to the index definition.");
    }
    else
    {
        Console.WriteLine($"Semantic configuration '{semanticConfigName}'
already exists in the index definition.");
    }
    index.SemanticSearch.DefaultConfigurationName = semanticConfigName;
}
}

```

2. Replace the search service URL with a valid endpoint.
3. Run the program.
4. Output is logged to a console window from `Console.WriteLine`. You should see messages for each step, including the JSON of the index schema with the new semantic configuration included.

Run semantic queries

In this section, the program runs several semantic queries in sequence.

1. Open the [QueryIndex project](#) in Visual Studio. The program consists of the following code.

This code uses a `SearchClient` for sending queries to an index.

C#

```
class SemanticQuery
{
    static async Task Main(string[] args)
    {
        string searchServiceName = "PUT-YOUR-SEARCH-SERVICE-NAME-HERE";
        string indexName = "hotels-sample-index";
        string endpoint = $"https://'{searchServiceName}'.search.windows.net";
        var credential = new Azure.Identity.DefaultAzureCredential();

        var client = new SearchClient(new Uri(endpoint), indexName,
        credential);

        // Query 1: Simple query
        string searchText = "walking distance to live music";
        Console.WriteLine("\nQuery 1: Simple query using the search string
'walking distance to live music'.");
        await RunQuery(client, searchText, new SearchOptions
        {
            Size = 5,
            QueryType = SearchQueryType.Simple,
            IncludeTotalCount = true,
            Select = { "HotelId", "HotelName", "Description" }
        });
        Console.WriteLine("Press Enter to continue to the next query...");
        Console.ReadLine();

        // Query 2: Semantic query (no captions, no answers)
        Console.WriteLine("\nQuery 2: Semantic query (no captions, no answers)");
        for ('walking distance to live music'.);
        var semanticOptions = new SearchOptions
        {
            Size = 5,
            QueryType = SearchQueryType.Semantic,
            SemanticSearch = new SemanticSearchOptions
            {
                SemanticConfigurationName = "semantic-config"
            },
            IncludeTotalCount = true,
            Select = { "HotelId", "HotelName", "Description" }
        };
        await RunQuery(client, searchText, semanticOptions);
        Console.WriteLine("Press Enter to continue to the next query...");
        Console.ReadLine();

        // Query 3: Semantic query with captions
        Console.WriteLine("\nQuery 3: Semantic query with captions.");
        var captionsOptions = new SearchOptions
        {
            Size = 5,
            QueryType = SearchQueryType.Semantic,
            SemanticSearch = new SemanticSearchOptions
            {
                SemanticConfigurationName = "semantic-config",
                QueryCaption = new QueryCaption(QueryCaptionType.Extractive)
            }
        };
    }
}
```

```

        {
            HighlightEnabled = true
        }
    },
    IncludeTotalCount = true,
    Select = { "HotelId", "HotelName", "Description" }
);
// Add the field(s) you want captions for to the QueryCaption.Fields
collection
captionsOptions.HighlightFields.Add("Description");
await RunQuery(client, searchText, captionsOptions, showCaptions:
true);
Console.WriteLine("Press Enter to continue to the next query...");
Console.ReadLine();

// Query 4: Semantic query with answers
// This query uses different search text designed for an answers
scenario
string searchText2 = "what's a good hotel for people who like to read";
searchText = searchText2; // Update searchText for the next query
Console.WriteLine("\nQuery 4: Semantic query with a verbatim answer
from the Description field for 'what's a good hotel for people who like to
read'.");
var answersOptions = new SearchOptions
{
    Size = 5,
    QueryType = SearchQueryType.Semantic,
    SemanticSearch = new SemanticSearchOptions
    {
        SemanticConfigurationName = "semantic-config",
        QueryAnswer = new QueryAnswer(QueryAnswerType.Extractive)
    },
    IncludeTotalCount = true,
    Select = { "HotelId", "HotelName", "Description" }
};
await RunQuery(client, searchText2, answersOptions, showAnswers: true);

static async Task RunQuery(
SearchClient client,
string searchText,
SearchOptions options,
bool showCaptions = false,
bool showAnswers = false)
{
    try
    {
        var response = await client.SearchAsync<SearchDocument>
(searchText, options);

        if (showAnswers && response.Value.SemanticSearch?.Answers !=
null)
        {
            Console.WriteLine("Extractive Answers:");
            foreach (var answer in
response.Value.SemanticSearch.Answers)

```

```
        {
            Console.WriteLine($"  {answer.Highlights}");
        }
        Console.WriteLine(new string('-', 40));
    }

    await foreach (var result in response.Value.GetResultsAsync())
    {
        var doc = result.Document;
        // Print captions first if available
        if (showCaptions && result.SemanticSearch?.Captions != null)
        {
            foreach (var caption in result.SemanticSearch.Captions)
            {
                Console.WriteLine($"Caption: {caption.Highlights}");
            }
            Console.WriteLine($"HotelId: {doc.GetString("HotelId")}");
            Console.WriteLine($"HotelName: {doc.GetString("HotelName")}");
            Console.WriteLine($"Description: {doc.GetString("Description")}");
            Console.WriteLine($"@search.score: {result.Score}");

            // Print @search.rerankerScore if available
            if (result.SemanticSearch != null && result.SemanticSearch.RerankerScore.HasValue)
            {
                Console.WriteLine($"@search.rerankerScore: {result.SemanticSearch.RerankerScore.Value}");
            }
            Console.WriteLine(new string('-', 40));
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error querying index: {ex.Message}");
    }
}
}
```

2. Replace the search service URL with a valid endpoint.
 3. Run the program.
 4. Output is logged to a console window from `Console.WriteLine`. You should see search results for each query.

Output for semantic query (no captions or answers)

This output is from the semantic query, with no captions or answers. The query string is 'walking distance to live music'.

Here, the initial results from the term query are rescored using the semantic ranking models. For this particular dataset and query, the first several results are in similar positions. The effects of semantic ranking are more pronounced in the remainder of the results.

Bash

```
HotelId: 24
HostName: Uptown Chic Hotel
Description: Chic hotel near the city. High-rise hotel in downtown, within walking
distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum
by day, and then head over to Benaroya Hall to catch the evening's concert
performance.
@search.score: 5.074317
@search.rerankerScore: 2.613231658935547
-----
HotelId: 2
HostName: Old Century Hotel
Description: The hotel is situated in a nineteenth century plaza, which has been
expanded and renovated to the highest architectural standards to create a modern,
functional and first-class hotel in which art and unique historical elements coexist
with the most modern comforts. The hotel also regularly hosts events like wine
tastings, beer dinners, and live music.
@search.score: 5.5153193
@search.rerankerScore: 2.271434783935547
-----
HotelId: 4
HostName: Sublime Palace Hotel
Description: Sublime Cliff Hotel is located in the heart of the historic center of
Sublime in an extremely vibrant and lively area within short walking distance to the
sites and landmarks of the city and is surrounded by the extraordinary beauty of
churches, buildings, shops and monuments. Sublime Cliff is part of a lovingly
restored 19th century resort, updated for every modern convenience.
@search.score: 4.8959594
@search.rerankerScore: 1.9861756563186646
-----
HotelId: 39
HostName: White Mountain Lodge & Suites
Description: Live amongst the trees in the heart of the forest. Hike along our
extensive trail system. Visit the Natural Hot Springs, or enjoy our signature hot
stone massage in the Cathedral of Firs. Relax in the meditation gardens, or join new
friends around the communal firepit. Weekend evening entertainment on the patio
features special guest musicians or poetry readings.
@search.score: 0.7334347
@search.rerankerScore: 1.9615401029586792
-----
HotelId: 15
HostName: By the Market Hotel
Description: Book now and Save up to 30%. Central location. Walking distance from
the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new
rooms. Impeccable service.
```

```
@search.score: 1.5502293  
@search.rerankerScore: 1.9085469245910645
```

```
-----  
Press Enter to continue to the next query...
```

Output for a semantic query with captions

Here are the results for the query that adds captions with hit highlighting.

```
Caption: Chic hotel near the city. High-rise hotel in downtown, within walking  
distance to<em> theaters, </em>art galleries, restaurants and shops. Visit<em>  
Seattle Art Museum </em>by day, and then head over to<em> Benaroya Hall </em>to  
catch the evening's concert performance.  
HotelId: 24  
HotelName: Uptown Chic Hotel  
Description: Chic hotel near the city. High-rise hotel in downtown, within walking  
distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum  
by day, and then head over to Benaroya Hall to catch the evening's concert  
performance.  
@search.score: 5.074317  
@search.rerankerScore: 2.613231658935547  
-----  
Caption:  
HotelId: 2  
HotelName: Old Century Hotel  
Description: The hotel is situated in a nineteenth century plaza, which has been  
expanded and renovated to the highest architectural standards to create a modern,  
functional and first-class hotel in which art and unique historical elements coexist  
with the most modern comforts. The hotel also regularly hosts events like wine  
tastings, beer dinners, and live music.  
@search.score: 5.5153193  
@search.rerankerScore: 2.271434783935547  
-----  
Caption: Sublime Cliff Hotel is located in the heart of the historic center of  
Sublime in an extremely vibrant and lively area within<em> short walking distance  
</em>to the sites and landmarks of the city and is surrounded by the extraordinary  
beauty of churches, buildings, shops and monuments. Sublime Cliff is part of a  
lovingly restored 19th century resort,.  
HotelId: 4  
HotelName: Sublime Palace Hotel  
Description: Sublime Cliff Hotel is located in the heart of the historic center of  
Sublime in an extremely vibrant and lively area within short walking distance to the  
sites and landmarks of the city and is surrounded by the extraordinary beauty of  
churches, buildings, shops and monuments. Sublime Cliff is part of a lovingly  
restored 19th century resort, updated for every modern convenience.  
@search.score: 4.8959594  
@search.rerankerScore: 1.9861756563186646  
-----  
Caption: Live amongst the trees in the heart of the forest. Hike along our extensive  
trail system. Visit the Natural Hot Springs, or enjoy our signature hot stone
```

massage in the Cathedral of Firs. Relax in the meditation gardens, or join new friends around the communal firepit. Weekend evening entertainment on the patio features special guest musicians or poetry readings.

HotelId: 39

HotelName: White Mountain Lodge & Suites

Description: Live amongst the trees in the heart of the forest. Hike along our extensive trail system. Visit the Natural Hot Springs, or enjoy our signature hot stone massage in the Cathedral of Firs. Relax in the meditation gardens, or join new friends around the communal firepit. Weekend evening entertainment on the patio features special guest musicians or poetry readings.

@search.score: 0.7334347

@search.rerankerScore: 1.9615401029586792

Caption: Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service.

HotelId: 15

HotelName: By the Market Hotel

Description: Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service.

@search.score: 1.5502293

@search.rerankerScore: 1.9085469245910645

Press Enter to continue to the next query...

Output for semantic answers

The final query returns a semantic answer. Notice that we changed the query string for this example: 'what's a good hotel for people who like to read'.

Semantic ranker can produce an answer to a query string that has the characteristics of a question. The generated answer is extracted verbatim from your content so it won't include composed content like what you might expect from a chat completion model. If the semantic answer isn't useful for your scenario, you can omit `semantic_answers` from your code.

To produce a semantic answer, the question and answer must be closely aligned, and the model must find content that clearly answers the question. If potential answers fail to meet a confidence threshold, the model doesn't return an answer. For demonstration purposes, the question in this example is designed to get a response so that you can see the syntax.

Recall that answers are *verbatim content* pulled from your index and might be missing phrases that a user would expect to see. To get *composed answers* as generated by a chat completion model, considering using a [RAG pattern](#) or [agentic retrieval](#).

Bash

Extractive Answers:

Nature is Home on the beach. Explore the shore by day, and then come home to our shared living space to relax around a stone fireplace, sip something warm, and explore the library by night. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackouts may apply.

HotelId: 1

HotelName: Stay-Kay City Hotel

Description: This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.

@search.score: 2.0361428

@search.rerankerScore: 2.124817371368408

HotelId: 16

HotelName: Double Sanctuary Resort

Description: 5 star Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Traveler magazine. Free WiFi, Flexible check in/out, Fitness Center & espresso in room.

@search.score: 3.759768

@search.rerankerScore: 2.0705394744873047

HotelId: 38

HotelName: Lakeside B & B

Description: Nature is Home on the beach. Explore the shore by day, and then come home to our shared living space to relax around a stone fireplace, sip something warm, and explore the library by night. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackouts may apply.

@search.score: 0.7308748

@search.rerankerScore: 2.041472911834717

HotelId: 2

HotelName: Old Century Hotel

Description: The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.

@search.score: 3.391012

@search.rerankerScore: 2.0231292247772217

HotelId: 15

HotelName: By the Market Hotel

Description: Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service.

@search.score: 1.3198771

@search.rerankerScore: 2.021622657775879

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

Related content

In this quickstart, you learned how to invoke semantic ranking on an existing index. We recommend trying semantic ranking on your own indexes as a next step. The following articles can help you get started.

- [Semantic ranking overview](#)
- [Configure semantic ranker](#)
- [Add query rewrite to semantic ranking](#)
- [Use scoring profiles and semantic ranking together](#)

Last updated on 11/20/2025

Quickstart: Chat with Azure OpenAI models using your own data

ⓘ Note

This document refers to the [Microsoft Foundry \(classic\)](#) portal.

 [View the Microsoft Foundry \(new\) documentation](#) to learn about the new portal.

In this quickstart, you use your own data with Azure OpenAI models to create a powerful, conversational AI platform that enables faster and more accurate communication.

ⓘ Important

There are new ways to build conversational solutions with your own data. For the latest recommended approach, see [Quickstart: Use agentic retrieval in Azure AI Search](#).

Prerequisites

- Download the example data from [GitHub](#) if you don't have your own data.

Add your data using Microsoft Foundry portal

💡 Tip

Alternatively, you can [use the Azure Developer CLI](#) to programmatically create the resources needed for Azure OpenAI On Your Data.

To add your data using the portal:

1. Sign in to [Microsoft Foundry](#). Make sure the **New Foundry** toggle is off. These steps refer to **Foundry (classic)**.

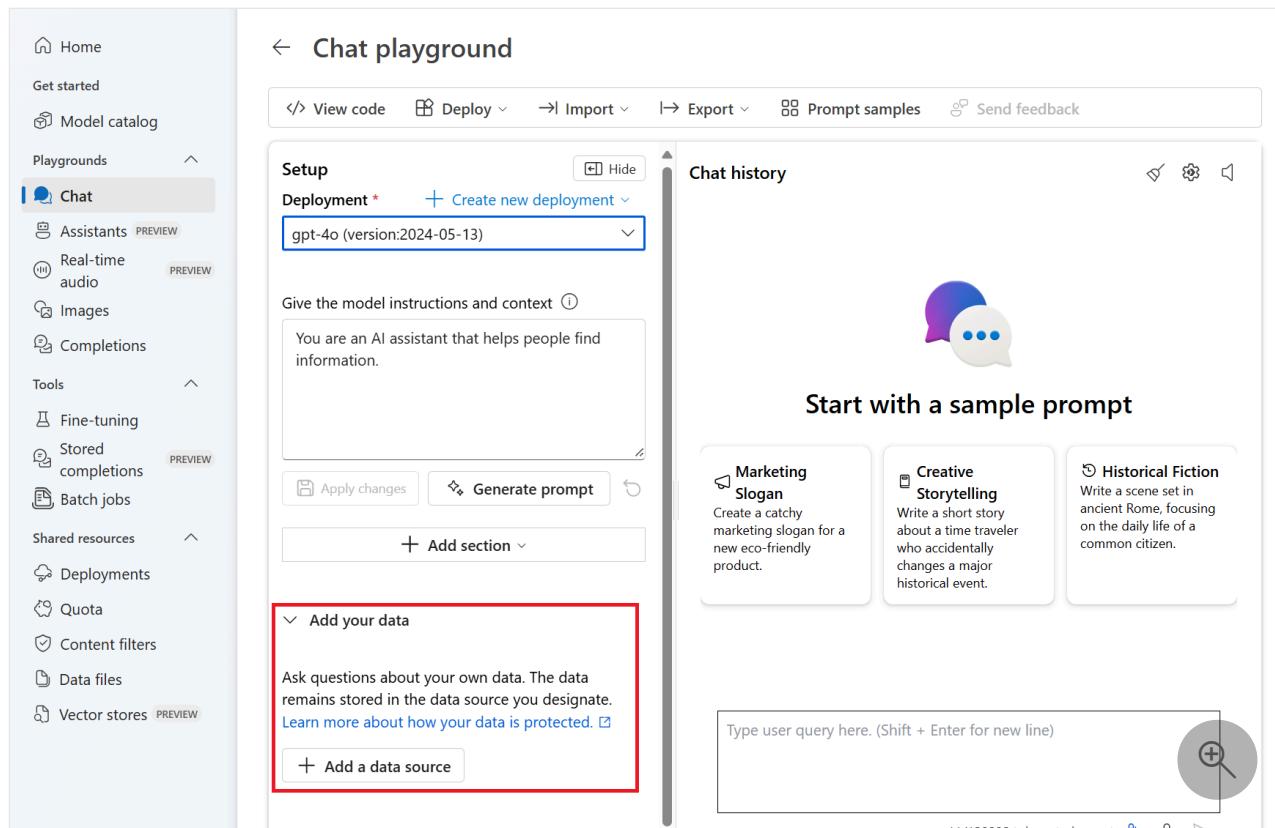


New Foundry

2. Select your Azure OpenAI resource. If you have a Foundry resource, you can [create a Foundry project](#).
3. From the left pane, select **Playgrounds > Chat**.

4. In the **Setup** pane, select your model deployment.

5. Select **Add your data > Add a data source**.



6. On the **Data source** page:

a. Under **Select data source**, select **Upload files (preview)**.

Tip

- This option requires an Azure Blob Storage resource and Azure AI Search resource to access and index your data. For more information, see [Data source options](#) and [Supported file types and formats](#).
- For documents and datasets with long text, we recommend that you use the [data preparation script](#).

b. [Cross-origin resource sharing](#) (CORS) is required for Azure OpenAI to access your storage account. If CORS isn't already enabled for your Azure Blob Storage resource, select **Turn on CORS**.

c. Select your Azure AI Search resource.

d. Enter a name for your new index.

e. Select the checkbox that acknowledges the billing effects of using Azure AI Search.

f. Select Next.

The screenshot shows the 'Add data' wizard with the title 'Select or add data source'. On the left, a sidebar lists steps: 'Data source' (selected), 'Upload files', 'Data management', and 'Review and finish'. The main area contains fields for 'Subscription' (dropdown), 'Select Azure Blob storage resource' (dropdown), 'Select Azure AI Search resource' (dropdown), 'Enter the index name' (text input), and a checkbox for 'Add vector search to this search resource'. At the bottom are 'Next' and 'Cancel' buttons.

7. On the **Upload files** page:

- Select **Browse for a file**, and then select your own data or the sample data you downloaded from the [prerequisites](#).
- Select **Upload files**.
- Select **Next**.

8. On the **Data management** page:

- Choose whether to enable [semantic search](#) or [vector search](#) for your index.

Important

- [Semantic search](#) and [vector search](#) are subject to additional pricing. Your Azure AI Search resource must be on the Basic tier or higher to enable semantic search or vector search. For more information, see [Choose a tier](#) and [Service limits](#).

- To help improve the quality of the information retrieval and model response, we recommend that you enable [semantic search](#) for the following data source languages: English, French, Spanish, Portuguese, Italian, Germany, Chinese(Zh), Japanese, Korean, Russian, and Arabic.

b. Select **Next**.

9. On the **Data connection** page:

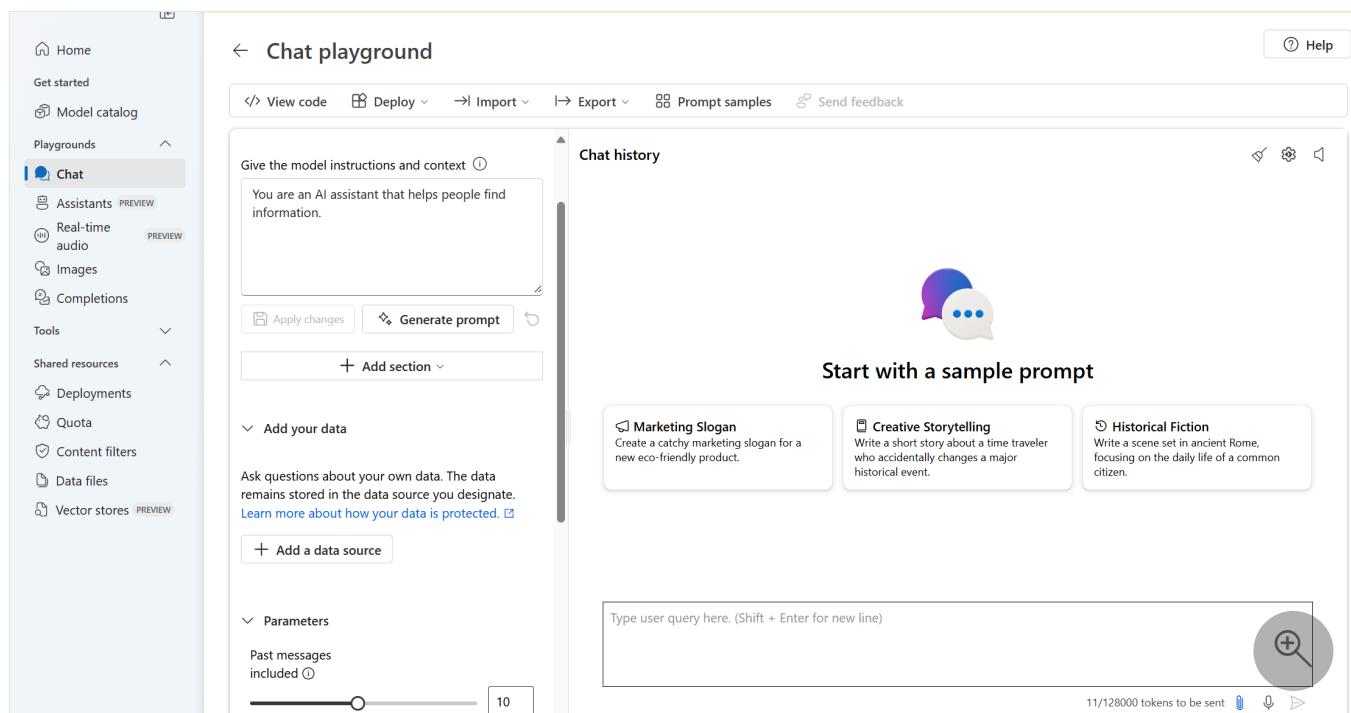
- Choose whether to authenticate using a **System assigned managed identity** or an **API key**.
- Select **Next**.

10. Review your configurations, and then select **Save and close**.

You can now chat with the model, which uses your data to construct the response.

Chat playground

Start exploring Azure OpenAI capabilities with a no-code approach through the chat playground. It's simply a text box where you can submit a prompt to generate a completion. From this page, you can quickly iterate and experiment with the capabilities.

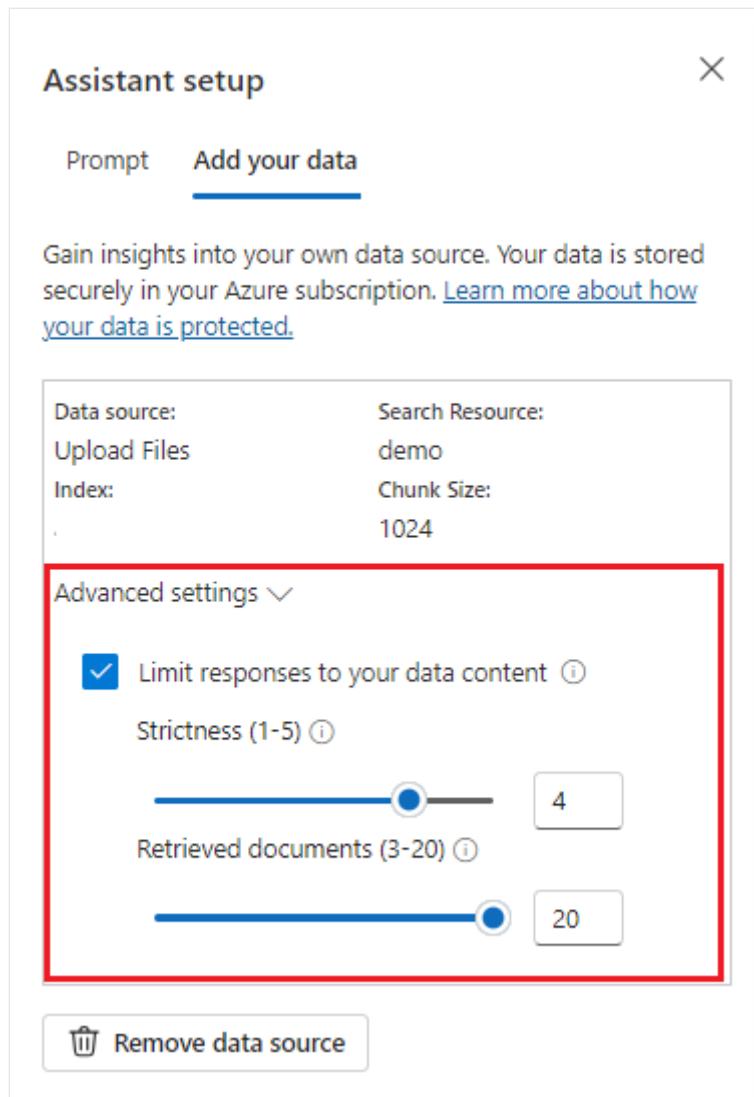


The playground gives you options to tailor your chat experience. On the top menu, you can select **Deploy** to determine which model generates a response using the search results from your index. You choose the number of past messages to include as conversation history for

future generated responses. [Conversation history](#) gives context to generate related responses but also consumes [token usage](#). The input token progress indicator keeps track of the token count of the question you submit.

The **Advanced settings** on the left are [runtime parameters](#), which give you control over retrieval and search relevant information from your data. A good use case is when you want to make sure responses are generated only based on your data or you find the model cannot generate a response based on existed information on your data.

- **Strictness** determines the system's aggressiveness in filtering search documents based on their similarity scores. Setting strictness to 5 indicates that the system will aggressively filter out documents, applying a very high similarity threshold. [Semantic search](#) can be helpful in this scenario because the ranking models do a better job of inferring the intent of the query. Lower levels of strictness produce more verbose answers, but might also include information that isn't in your index. This is set to 3 by default.
- **Retrieved documents** is an integer that can be set to 3, 5, 10, or 20, and controls the number of document chunks provided to the large language model for formulating the final response. By default, this is set to 5.
- When **Limit responses to your data** is enabled, the model attempts to only rely on your documents for responses. This is set to true by default.



Send your first query. The chat models perform best in question and answer exercises. For example, "*What are my available health plans?*" or "*What is the health plus option?*".

Queries that require data analysis would probably fail, such as "*Which health plan is most popular?*". Queries that require information about all of your data will also likely fail, such as "*How many documents have I uploaded?*". Remember that the search engine looks for chunks having exact or similar terms, phrases, or construction to the query. And while the model might understand the question, if search results are chunks from the data set, it's not the right information to answer that kind of question.

Chats are constrained by the number of documents (chunks) returned in the response (limited to 3-20 in Foundry portal playground). As you can imagine, posing a question about "all of the titles" requires a full scan of the entire vector store.

Deploy your model

Once you're satisfied with the experience, you can deploy a web app directly from the portal by selecting the **Deploy to** button.

← Chat playground

The screenshot shows the Microsoft Foundry Models Chat playground interface. At the top, there's a navigation bar with 'View code', 'Deploy' (with a dropdown menu showing '...as a web app'), 'Import', and 'Export'. Below the navigation is a text input field containing the placeholder 'Give the model instructions and context'. A tooltip above the input field says '...as a web app'. To the right of the input field is a button with a magnifying glass icon. The main area contains the text 'You are an AI assistant that helps people find'.

This gives you the option to either deploy to a standalone web application, or a copilot in Copilot Studio (preview) if you're [using your own data](#) on the model.

As an example, if you choose to deploy a web app:

The first time you deploy a web app, you should select **Create a new web app**. Choose a name for the app, which will become part of the app URL. For example,

`https://<appname>.azurewebsites.net`.

Select your subscription, resource group, location, and pricing plan for the published app. To update an existing app, select **Publish to an existing web app** and choose the name of your previous app from the dropdown menu.

If you choose to deploy a web app, see the [important considerations](#) for using it.

Clean up resources

If you want to clean up and remove an Azure OpenAI or Azure AI Search resource, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- [Azure AI Search resources](#)
- [Azure app service resources](#)

Next steps

- Learn more about [using your data in Azure OpenAI in Microsoft Foundry Models](#)
- [Chat app sample code on GitHub ↗](#)

Last updated on 11/18/2025

Quickstart: Use agentic retrieval in the Azure portal

!Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this quickstart, you use [agentic retrieval](#) in the Azure portal to create a conversational search experience powered by documents indexed in Azure AI Search and large language models (LLMs) from Azure OpenAI in Foundry Models.

The portal guides you through the process of creating the following objects:

- A *knowledge source* that references a container in Azure Blob Storage. When you create a blob knowledge source, Azure AI Search automatically generates an index and other pipeline objects to ingest and enrich your content for agentic retrieval.
- A *knowledge base* that uses agentic retrieval to infer the underlying information need, plan and execute subqueries, and formulate a natural-language answer using the optional answer synthesis output mode.

Afterwards, you test the knowledge base by submitting a complex query that requires information from multiple documents and reviewing the synthesized answer.

!Important

Because the portal uses the 2025-08-01-preview REST API version for agentic retrieval, the knowledge source and knowledge base created in this quickstart aren't compatible with the latest 2025-11-01-preview. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#) in any region that provides agentic retrieval.
- An [Azure Blob Storage account](#).

- A Microsoft Foundry project and resource. When you create a project, the resource is automatically created.
- For text-to-vector conversion, an embedding model [deployed to your project](#). You can use any `text-embedding` model, such as `text-embedding-3-large`.
- For query planning and answer generation, an LLM [deployed to your project](#). You can use any [portal-supported LLM](#).

Supported LLMs

Although agentic retrieval [programmatically supports several LLMs](#), the portal currently supports the following LLMs:

- `gpt-4o`
- `gpt-4o-mini`
- `gpt-5`
- `gpt-5-mini`
- `gpt-5-nano`

Configure access

Before you begin, make sure you have permissions to access content and operations. We recommend Microsoft Entra ID for authentication and role-based access for authorization. You must be an **Owner** or **User Access Administrator** to assign roles. If roles aren't feasible, use [key-based authentication](#) instead.

To configure access for this quickstart, select each of the following tabs.

Azure AI Search

Azure AI Search provides the agentic retrieval pipeline. Configure access for yourself and your search service to read and write data, interact with other Azure services, and run the pipeline.

On your Azure AI Search service:

1. [Enable role-based access](#).
2. [Configure a system-assigned managed identity](#).
3. [Assign the following roles](#) to yourself.

- Search Service Contributor
- Search Index Data Contributor
- Search Index Data Reader

Important

Agentic retrieval has two token-based billing models:

- Billing from Azure AI Search for agentic retrieval.
- Billing from Azure OpenAI for query planning and answer synthesis.

For more information, see [Availability and pricing of agentic retrieval](#).

Prepare sample data

This quickstart uses sample JSON documents from NASA's Earth at Night e-book, but you can also use your own files. The documents describe general science topics and images of Earth at night as observed from space.

To prepare the sample data for this quickstart:

1. Sign in to the [Azure portal](#) and select your Azure Blob Storage account.
2. From the left pane, select **Data storage > Containers**.
3. Create a container named **earth-at-night-data**.
4. Upload the [sample JSON documents](#) to the container.

Create a knowledge source

A knowledge source is a reusable reference to your source data. In this section, you create a [blob knowledge source](#), which triggers the creation of a *data source*, *skillset*, *index*, and *indexer* to automate data indexing and enrichment. You review these objects in a later section.

You also configure a *vectorizer*, which uses your deployed embedding model to convert text into vectors and match documents based on semantic similarity. The vectorizer, vector fields, and vectors will be added to the auto-generated index.

To create the knowledge source for this quickstart:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Knowledge retrieval > Knowledge sources**.
3. Select **Add knowledge source > Add knowledge source**.
4. Enter **earth-at-night-ks** for the name.
5. Select **Azure blob**, and then select your subscription, storage account, and container with the sample data.
6. Select the **Authenticate using managed identity** checkbox. Leave the identity type as **System-assigned**.
7. Select **Add vectorizer**.
8. Select **Microsoft Foundry** for the kind, and then select your subscription, project, and embedding model deployment.
9. Select **System managed identity** for the authentication type.
10. Create the knowledge source.

Configurations

X

earth-at-night-ks

Subscription *

my-subscription



Storage account *

mystorageaccount



Blob container *

earth-at-night-data



Blob folder

your/folder/here

Authenticate using managed identity. [Learn more](#)

Managed identity type

System-assigned



Schedule

Daily



Vectorizer



vectorizer-1762198899495



Deployment: text-embedding-ada-002

Model: text-embedding-ada-002

Resource: <https://myresource.openai.azure.com/>

[Create](#)

[Back](#)



Create a knowledge base

⚠ Note

The portal uses the 2025-08-01-preview, which refers to "knowledge bases" as "knowledge agents." Although the portal UI uses the latest terminology, the underlying REST API objects and JSON payloads still use "knowledge agents."

A knowledge base uses your knowledge source and deployed LLM to orchestrate agentic retrieval. When a user submits a complex query, the LLM generates subqueries that are sent simultaneously to your knowledge source. Azure AI Search then semantically ranks the results for relevance and combines the best results into a single, unified response.

The output mode determines how the knowledge base formulates answers. You can either use extractive data for verbatim content or [answer synthesis](#) for natural-language answer generation. By default, the portal uses answer synthesis.

To create the knowledge base for this quickstart:

1. From the left pane, select **Knowledge retrieval > Knowledge bases**.
2. Select **Add knowledge base > Add knowledge base**.
3. Enter **earth-at-night-kb** for the name.
4. Under **Model deployment**, select **Add model deployment**.
5. Select **Foundry** for the kind, and then select your subscription, project, and LLM deployment.
6. Select **System assigned identity** for the authentication type.
7. Save the model deployment.
8. Under **Knowledge sources**, select **earth-at-night-ks**.
9. Create the knowledge base.

 Create

Name *

earth-at-night-kb

Description

Describe your knowledge base

Model deployment *

 gpt-5-nano (myproject)

Knowledge sources *

1 knowledge source selected 

[Create new knowledge source](#)



earth-at-night-ks



Test agentic retrieval

The portal provides a chat playground where you can submit `retrieve` requests to the knowledge base, whose responses include references to your knowledge sources and debug information about the retrieval process.

To query the knowledge base:

1. Use the chat box to send the following query.

Why do suburban belts display larger December brightening than urban cores even though absolute light levels are higher downtown? Why is the Phoenix nighttime

street grid is so sharply visible from space, whereas large stretches of the interstate between midwestern cities remain comparatively dim?

2. Review the synthesized, citation-backed answer, which should be similar to the following example.

The suburban belts exhibit larger December brightening compared to urban cores due to the increased use of decorative and festive lighting in residential areas, which are more prevalent in suburban regions. In contrast, urban cores, despite having higher absolute light levels, experience less seasonal variation in lighting. The Phoenix nighttime street grid is sharply visible from space because of its regular grid layout and the extensive use of street lighting, which creates a consistent and bright pattern. Conversely, large stretches of interstate highways between Midwestern cities are less illuminated, as they primarily serve as transit routes with minimal lighting infrastructure, resulting in comparatively dim visibility from space.

3. Select the debug icon to review the activity log, which should be similar to the following example.

```
[  
 {  
   "type": "modelQueryPlanning",  
   "id": 0,  
   "inputTokens": 2081,  
   "outputTokens": 128,  
   "elapsedMs": 1577  
 },  
 {  
   "type": "azureBlob",  
   "id": 1,  
   "knowledgeSourceName": "earth-at-night-ks",  
   "queryTime": "2025-11-03T15:09:28.172Z",  
   "count": 0,  
   "elapsedMs": 731,  
   "azureBlobArguments": {  
     "search": "Why do suburban belts display larger December brightening than urban cores despite higher downtown light levels?"  
   }  
 },  
 {  
   "type": "azureBlob",  
   "id": 2,  
   "knowledgeSourceName": "earth-at-night-ks",  
   "queryTime": "2025-11-03T15:09:28.669Z",  
   "count": 3,  
   "elapsedMs": 497,  
   "azureBlobArguments": {
```

```
        "search": "Why is the Phoenix nighttime street grid sharply visible from space compared to dim interstates in the Midwest?"  
    }  
,  
{  
    "type": "semanticReranker",  
    "id": 3,  
    "inputTokens": 0  
,  
{  
    "type": "modelAnswerSynthesis",  
    "id": 4,  
    "inputTokens": 3938,  
    "outputTokens": 136,  
    "elapsedMs": 1963  
}  
]  
]
```

The activity log offers insight into the steps taken during retrieval, including query planning and execution, semantic ranking, and answer synthesis. For more information, see [Review the activity array](#).

Review the created objects

Azure AI Search automatically generates a data source, skillset, index, and indexer for each blob knowledge source. These objects form an end-to-end pipeline for data ingestion, enrichment, chunking, and vectorization. You can review these objects to learn how your data is processed for agentic retrieval.

To review the auto-generated objects:

1. From the left pane, select **Search management**.
2. Check the data source to verify the connection to your blob storage container.
3. Check the skillset to see how your content is chunked and vectorized using your embedding model.
4. Check the index to see how your content is indexed and exposed for retrieval, including which fields are searchable and filterable and which fields store vectors for similarity search.
5. Check the indexer for success or failure messages. Connection or quota errors appear here.

Clean up resources

When you work in your own subscription, it's a good idea to finish a project by determining whether you still need the resources you created. Resources that are left running can cost you money.

In the Azure portal, you can manage your Azure AI Search, Azure Blob Storage, and Foundry resources by selecting **All resources** or **Resource groups** from the left pane.

You can also delete the knowledge source and knowledge base on their respective portal pages. When you delete the knowledge source, the portal prompts you to delete the associated data source, skillset, index, and indexer.

Next step

[Learn more about agentic retrieval](#)

Last updated on 11/18/2025

Quickstart: Create a search index in the Azure portal

Important

The **Import data (new)** wizard now supports keyword search, which was previously only available in the **Import data** wizard. We recommend the new wizard for an improved search experience. For more information about how we're consolidating the wizards, see [Import data wizards in the Azure portal](#).

In this quickstart, you use the **Import data (new)** wizard and sample data about fictitious hotels to create your first search index. The wizard requires no code to create an index, helping you write interesting queries within minutes.

The wizard creates multiple objects on your search service, including a searchable [index](#), an [indexer](#), and a data source connection for automated data retrieval. At the end of this quickstart, you review each object.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription. You can use a free service for this quickstart.
- An [Azure Storage account](#). Use Azure Blob Storage or Azure Data Lake Storage Gen2 (storage account with a hierarchical namespace) on a standard performance (general-purpose v2) account. To avoid bandwidth charges, use the same region as Azure AI Search.

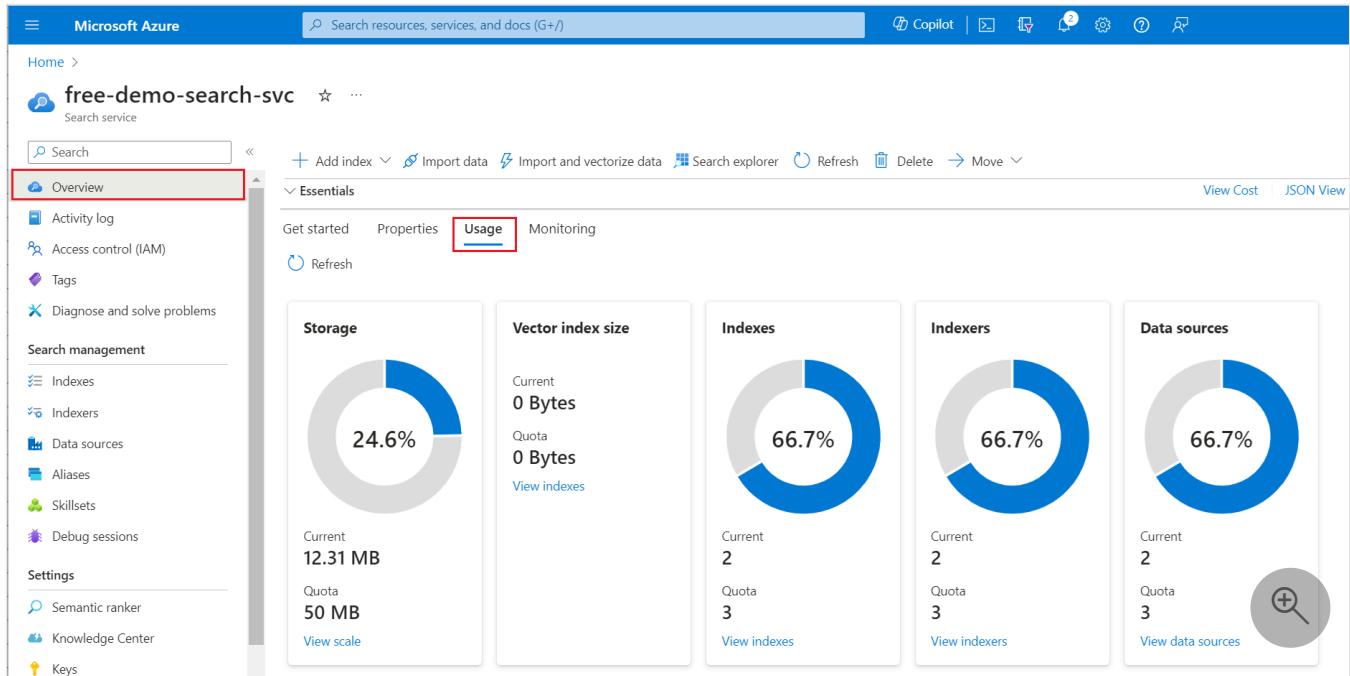
Check for network access

For this quickstart, all of the preceding resources must have public access enabled so that the Azure portal nodes can access them. Otherwise, the wizard fails. After the wizard runs, you can enable firewalls and private endpoints on the integration components for security. For more information, see [Secure connections in the import wizards](#).

Check for space

Many customers start with a free search service, which is limited to three indexes, three indexers, and three data sources. This quickstart creates one of each, so before you begin, make sure you have room for extra objects.

On the **Overview** page, select **Usage** to see how many indexes, indexers, and data sources you currently have.



Prepare sample data

This quickstart uses a JSON document that contains metadata for 50 fictitious hotels, but you can also use your own files.

To prepare the sample data for this quickstart:

1. Download the [sample JSON document](#).
2. Sign in to the [Azure portal](#) and select your Azure Storage account.
3. From the left pane, select **Data storage > Containers**.
4. Create a container named **hotels-sample**.
5. Upload the **HotelsData_toAzureBlobs.json** file to the container.

Start the wizard

To start the wizard for this quickstart:

1. Sign in to the [Azure portal](#) and select your search service.

2. On the **Overview** page, select **Import data (new)**.



3. Select your data source: **Azure Blob Storage** or **Azure Data Lake Storage Gen2**.

A screenshot of the 'Select your data source' interface. It shows a grid of data source options. The first two options, 'Azure Blob Storage' and 'Azure Data Lake Storage Gen2', are highlighted with red boxes. Other options include: Azure SQL Database, Azure Table Storage, Azure Cosmos DB, SharePoint, OneDrive, OneDrive for Business, Azure File Storage, Azure Queues, Service Bus, Amazon S3, Dropbox, and SFTP - SSH. A search bar at the top says 'Let's start by picking a data source...' and 'Available options are listed below. Learn more'. A magnifying glass icon is in the bottom right corner.

4. Select **Keyword search**.

A screenshot of the 'Select Keyword search' interface. It shows three scenarios: 'Keyword search' (highlighted with a red box), 'RAG', and 'Multimodal RAG'. Each scenario has a description and an icon. The 'Keyword search' box says: 'Ingest text for keyword search. Optionally, add AI skills to extract, infer, or create new searchable content.' The 'RAG' box says: 'Ingest text and simple images containing text (via OCR) to enable AI-powered answers.' The 'Multimodal RAG' box says: 'Ingest text and complex images (diagrams, charts, workflows) where interpreting visual elements is necessary for insights.' A magnifying glass icon is in the bottom right corner.

Create and load a search index

In this section, you create and load an index in five steps.

Connect to a data source

Azure AI Search requires a connection to a data source for content ingestion and indexing. In this case, the data source is your Azure Storage account.

To connect to the sample data:

1. On the **Connect to your data** page, select your Azure subscription.

2. Select your storage account, and then select the **hotels-sample** container.

3. Select **JSON array** for the parsing mode.

Configure your Azure Blob Storage

Connect to Azure Blob Storage to access your structured and unstructured data files, including PDFs. [Learn more](#)

Subscription *	my-subscription
Storage account *	mystorageaccount
Blob container * ⓘ	hotels-sample
Blob folder ⓘ	your/folder/here
Parsing mode	JSON array
Document Root ⓘ	/level1/level2

Enable deletion tracking. ⓘ

Authenticate using managed identity. [Learn more](#)



4. Select **Next**.

Skip configuration for skills

The wizard supports skillset creation and **AI enrichment** during indexing, which are beyond the scope of this quickstart. Skip this step by selecting **Next**.

💡 Tip

For a similar walkthrough that focuses on AI enrichment, see [Quickstart: Create a skillset in the Azure portal](#).

Configure the index

Based on the structure and content of the sample hotel data, the wizard infers a schema for your search index.

To configure the index:

1. For each of the following fields, select **Configure field**, and then set the respective attributes.

[Expand table](#)

Fields	Attributes
HotelId	Key, Retrievable, Filterable, Sortable, Searchable
HostName, Category	Retrievable, Filterable, Sortable, Searchable
Description, Description_fr	Retrievable, Searchable
Tags	Retrievable, Filterable, Searchable
ParkingIncluded, IsDeleted, Location	Retrievable, Filterable, Facetable
LastRenovationDate, Rating	Retrievable, Filterable, Sortable
Address.StreetAddress, Rooms.Description, Rooms.Description_fr	Retrievable, Searchable
Address.City, Address.StateProvince, Address.PostalCode, Address.Country	Retrievable, Filterable, Facetable, Searchable, Sortable
Rooms.Type, Rooms.BedOptions, Rooms.Tags	Retrievable, Filterable, Facetable, Searchable
Rooms.BaseRate, Rooms.SleepsCount, Rooms.SmokingAllowed	Retrievable, Filterable, Facetable

Preview index fields

Review the fields that will be included in your index. Index fields can obtain values from your data source and from skill outputs if you added enrichments in the previous step. [Learn more](#)

+ Add field + Add subfield ⚙ Configure field ⏪ Reset 🗑 Delete Hide auto-generated fields

Source column	Target index field name	Target index field type
metadata_storage_name	title	Edm.String
metadata_storage_path	id	Edm.String
HotelId	HotelId	Edm.String
HostName	HostName	Edm.String
Description	Description	Edm.String
Description_fr	Description_fr	Edm.String
Category	Category	Edm.String
Tags	Tags	Collection(Edm.String)
ParkingIncluded	ParkingIncluded	Edm.Boolean
IsDeleted	IsDeleted	Edm.Boolean
LastRenovationDate	LastRenovationDate	Edm.DateTimeOffset
Rating	Rating	Edm.Double
Address	Address	Edm.ComplexType
StreetAddress	StreetAddress	Edm.String
City	City	Edm.String
StateProvince	StateProvince	Edm.String
PostalCode	PostalCode	Edm.String
Country	Country	Edm.String
Location	Location	Edm.GeographyPoint

2. Select Next.

At a minimum, the index requires a name and a collection of fields. The wizard scans for unique string fields and marks one as the document key, which uniquely identifies each document in the index.

Each field has a name, [data type](#), and attributes that control how the field is used in the index. You can enable or disable the following attributes:

[Expand table

Attribute	Description	Applicable data types
Retrievable	Fields returned in a query response.	Strings and integers
Filterable	Fields that accept a filter expression.	Strings and integers
Sortable	Fields that accept an orderby expression.	Strings and integers
Facetable	Fields used in a faceted navigation structure.	Strings and integers
Searchable	Fields used in full-text search. Strings are searchable, but numeric and Boolean fields are often marked as not searchable.	Strings

Attributes affect storage in different ways. For example, filterable fields consume extra storage, while retrievable fields don't. For more information about attributes and data types, see [Configure field definitions](#).

If you want autocomplete or suggested queries, specify [Suggesters](#).

Skip advanced settings

The wizard offers advanced settings for semantic ranking and index scheduling, which are beyond the scope of this quickstart. Skip this step by selecting [Next](#).

Review and create the objects

The last step is to review your configuration and create the index, indexer, and data source on your search service. The indexer automates the process of extracting content from your data source and loading it into the index, enabling keyword search.

To review and create the objects:

1. Change the object name prefix to **hotels-sample**.

2. Review the object configurations.

Based on your configuration, the wizard will create an index, an indexer, a data source, and a skillset on your search service. You can view and manage these objects after they're created, but their names and many other properties are fixed. To customize the name, change the object name prefix.

Objects name prefix

Review your configuration

Data source	Azure Blob Storage - hotels-sample
AI enrichments	Disabled
Semantic ranker	Enabled
Indexer run schedule	Once



AI enrichment, semantic ranker, and indexer scheduling are either disabled or set to their default values because you skipped their wizard steps.

3. Select **Create** to simultaneously create the objects and run the indexer.

Monitor indexer progress

You can monitor the creation of the indexer and index in the Azure portal. The **Overview** page provides links to the objects created on your search service.

To monitor the progress of the indexer:

1. From the left pane, select **Indexes**.
2. Find **hotels-sample-indexer** in the list.

Home > free-demo-search-svc

 **free-demo-search-svc** | Indexers  

Search service

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Search management

Indexes Indexers Data sources Aliases Skillsets Debug sessions

Add indexer Refresh Delete

Filter by name...

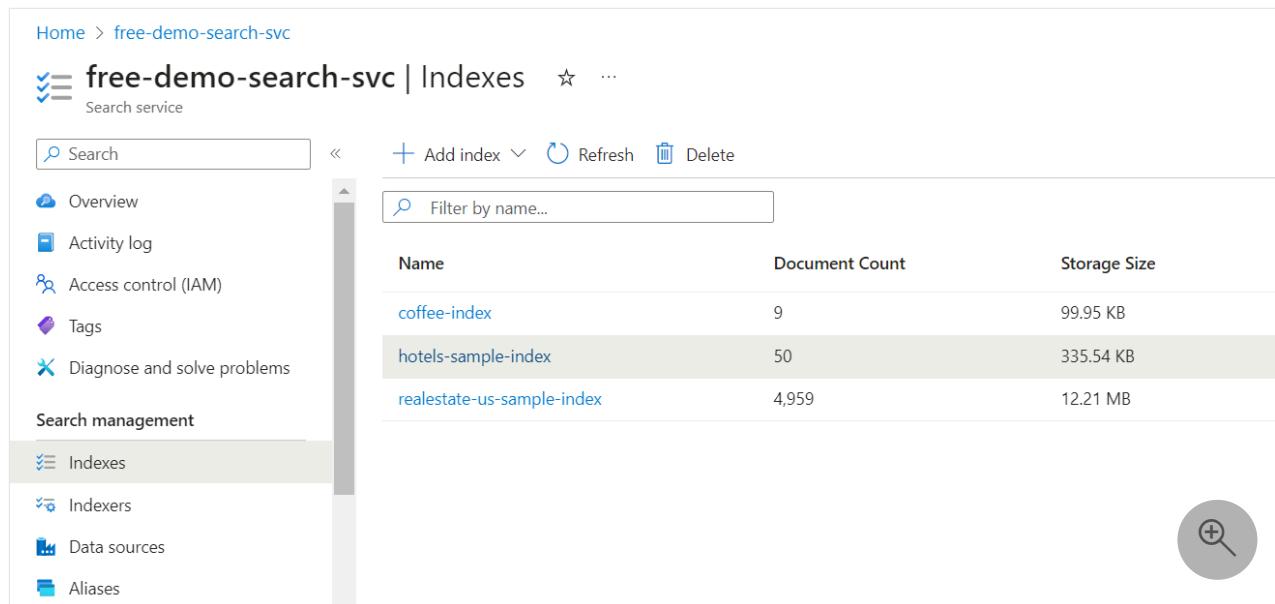
Status	Name	Last run	Docs succeeded	Errors/Warnings
Success	coffee-indexer	1 year ago	9/9	0/0
In progress	hotels-sample-indexer	2 seconds ago	0/0	0/0
Success	realestate-us-sample-in...	3 hours ago	4959/4959	0/0



It can take a few minutes for the results to update. You should see the newly created indexer with a status of **In progress** or **Success**. The list also shows the number of documents indexed.

Check search index results

1. From the left pane, select **Indexes**.
2. Select **hotels-sample-index**. If the index has zero documents or storage, wait for the Azure portal to refresh.



The screenshot shows the Azure portal interface for a search service named "free-demo-search-svc". The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Search management, Indexes (which is selected and highlighted in grey), Indexers, Data sources, and Aliases. The main content area displays the "Indexes" page with a table showing three entries:

Name	Document Count	Storage Size
coffee-index	9	99.95 KB
hotels-sample-index	50	335.54 KB
realestate-us-sample-index	4,959	12.21 MB

A search bar at the top allows filtering by name. A large circular button with a magnifying glass icon is located in the bottom right corner of the main content area.

3. Select the **Fields** tab to view the index schema.
4. Check which fields are **Filterable** or **Sortable** so that you know what queries to write.

hotels-sample-index ...

X

[Save](#) [Discard](#) [Refresh](#) [Create Demo App](#) [Edit JSON](#) [Delete](#)
Documents i Storage i

50 335.54 KB

Search explorer Fields CORS Scoring profiles Vector profiles

[+ Add field](#) [+ Add subfield](#) [Delete](#) [Autocomplete settings](#)
 Search field names

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable
		<input type="checkbox"/>				
HotelId	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
HotelName	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description_fr	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Category	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tags	StringCollection	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ParkingIncluded	Boolean	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
LastRenovationDate	DateTimeOffset	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Rating	Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
> Address	ComplexType					
Location	GeographyPoint	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		



Add or change fields

On the **Fields** tab, you can create a field by selecting **Add field** and specifying a name, [supported data type](#), and attributes.

Changing existing fields is more difficult. Existing fields have a physical representation in the search index, so they aren't modifiable, not even in code. To fundamentally change an existing field, you must create a new field to replace the original. You can add other constructs, such as scoring profiles and CORS options, to an index at any time.

Review the index definition options to understand what you can and can't edit during index design. If an option appears dimmed, you can't modify or delete it.

Query with Search explorer

You now have a search index that can be queried using [Search explorer](#), which sends REST calls that conform to [Documents - Search Post \(REST API\)](#). This tool supports [simple query syntax](#) and [full Lucene query syntax](#) for keyword search.

To query your search index:

1. On the [Search explorer](#) tab, enter text to search on.

Home > free-demo-search-svc | Indexes >

hotels-sample-index ...

Save Discard Refresh Create Demo App Edit JSON Delete

Documents ① Storage ①

50 335.7 KB

Search explorer Fields CORS Scoring profiles Vector profiles

Query options View

new york hotel with pool or gym

Results

```
1 {  
2   "@odata.context": "https://free-demo-search-svc.search.windows.net/indexes('hotels-sample-index')/$metadata#hotels/_entity",  
3   "value": [  
4     {  
5       "@search.score": 22.319506,  
6       "HotelId": "1",  
7       "HotelName": "Secret Point Motel",  
8       "Description": "The hotel is ideally located on the main commercial artery of the city in the heart of New York City.",  
9       "Description_fr": "L'hôtel est idéalement situé sur la principale artère commerciale de la ville en plein cœur de New York.",  
10      "Category": "boutique",  
11      "Tags": [  
12        "pool",  
13        "air conditioning",  
14        "concierge"  
15      ],  
16    }]
```

2. To jump to nonvisible areas of the output, use the mini map.

Results

Mini-map

```
1 {  
2   "@odata.context": "https://my-cognitive-search-westus2.search.windows.net/indexes('hotels-sample-index')/$metadata#hotels/_entity",  
3   "value": [  
4     {  
5       "@search.score": 1.1639717,  
6       "HotelId": "38",  
7       "HotelName": "Lady Of The Lake B & B",  
8       "Description": "Nature is Home on the beach. Save up to 30 percent. Valid Nov 15 - Dec 15.",  
9       "Description_fr": "La nature est à la maison sur la plage. Economisez jusqu'à 30 % du 15 novembre au 15 décembre.",  
10      "Category": "luxury",  
11      "Tags": [  
12        "laundry service",  
13        "concierge",  
14        "view"  
15      ]  
16    }]
```

3. To specify syntax, switch to the JSON view.

new york hotel with pool or gym

Query view

JSON view

Results

```

1  {
2    "@odata.context": "https://free-demo-search-svc.search.windows.net/indexes('hotels-sample-index')/$metadata#de
3    "value": [
4      {
5        "@search.score": 22.319506,
6        "HotelId": "1",
7        "HotelName": "Secret Point Motel",
8        "Description": "The hotel is ideally located on the main commercial artery of the city in the heart of New York City. It is close to Times Square, Central Park, and many other landmarks. The hotel features a swimming pool, a fitness center, and a restaurant. The rooms are spacious and comfortable, with modern amenities like free Wi-Fi and air conditioning. The hotel is perfect for business travelers and tourists alike.", "Description_fr": "L'hôtel est idéalement situé sur la principale artère commerciale de la ville en plein cœur de New York. Il est à proximité de Times Square, du Central Park et d'autres attractions majeures. L'hôtel offre une piscine, un centre de fitness et un restaurant. Les chambres sont spacieuses et confortables, avec des équipements modernes comme le Wi-Fi gratuit et l'air conditionné. L'hôtel est idéal pour les voyageurs d'affaires et les touristes.", "Category": "Boutique",

```



Example queries for hotels-sample index

The following examples assume the JSON view and the latest preview REST API version.

Tip

The JSON view supports intellisense for parameter name completion. Place your cursor inside the JSON view and enter a space character to see a list of all query parameters. You can also enter a letter, like `s`, to see only the query parameters that begin with that letter.

Intellisense doesn't exclude invalid parameters, so use your best judgment.

Filter examples

Parking, tags, renovation date, rating, and location are filterable.

JSON

```
{
  "search": "beach OR spa",
  "select": "HotelId, HotelName, Description, Rating",
  "count": true,
  "top": 10,
  "filter": "Rating gt 4"
}
```

Boolean filters assume "true" by default.

JSON

```
{
  "search": "beach OR spa",
  "select": "HotelId, HotelName, Description, Rating",
  "count": true,
```

```
"top": 10,  
"filter": "ParkingIncluded"  
}
```

Geospatial search is filter based. The `geo.distance` function filters all results for positional data based on the specified `Location` and `geography'POINT` coordinates. The query seeks hotels within five kilometers of the latitude and longitude coordinates `-122.12 47.67`, which is "Redmond, Washington, USA." The query displays the total number of matches `&$count=true` with the hotel names and address locations.

JSON

```
{  
  "search": "*",  
  "select": "HotelName, Address/City, Address/StateProvince",  
  "count": true,  
  "top": 10,  
  "filter": "geo.distance(Location, geography'POINT(-122.12 47.67)') le 5"  
}
```

Full Lucene syntax examples

The default syntax is [simple syntax](#), but if you want fuzzy search, term boosting, or regular expressions, specify the [full syntax](#).

JSON

```
{  
  "queryType": "full",  
  "search": "seattle~",  
  "select": "HotelId, HotelName,Address/City, Address/StateProvince",  
  "count": true  
}
```

Misspelled query terms, like `seattle` instead of `Seattle`, don't return matches in a typical search. The `queryType=full` parameter invokes the full Lucene query parser, which supports the tilde (`~`) operand. When you use these parameters, the query performs a fuzzy search for the specified keyword and matches on terms that are similar but not an exact match.

Take a minute to try these example queries on your index. For more information, see [Querying in Azure AI Search](#).

Clean up resources

When you work in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

In the Azure portal, you can find and manage resources by selecting **All resources** or **Resource groups** from the left pane.

 **Note**

If you're using a free search service, remember that the limit is three indexes, three indexers, and three data sources. You can delete individual objects in the Azure portal to stay under the limit.

Next step

Try an Azure portal wizard to generate a ready-to-use web app that runs in a browser. Use this wizard on the small index you created in this quickstart, or use [sample data](#) for a richer search experience.

[Quickstart: Create a demo search app in the Azure portal](#)

Last updated on 12/05/2025

Quickstart: Vectorize text in the Azure portal

In this quickstart, you use the [Import data \(new\)](#) wizard in the Azure portal to get started with [integrated vectorization](#). The wizard chunks your content and calls an embedding model to vectorize the chunks at indexing and query time.

This quickstart uses text-based PDFs from the [azure-search-sample-data](#) repo. However, you can use images and still complete this quickstart.

Prerequisites

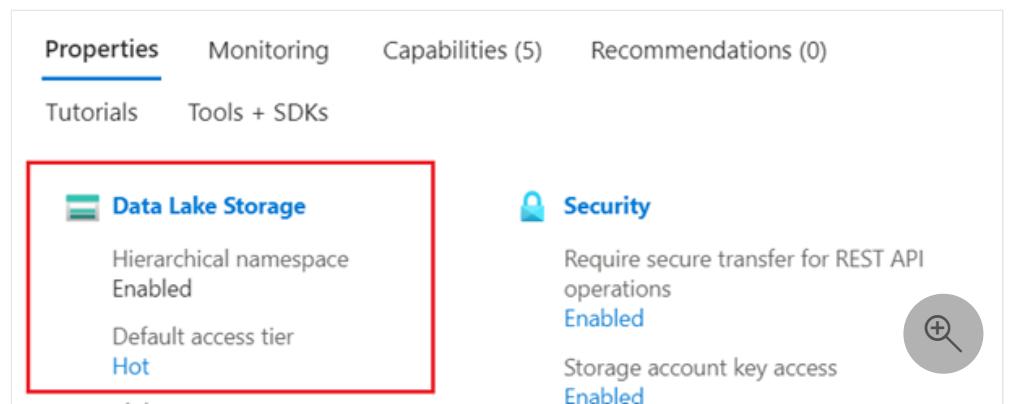
- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#). We recommend the Basic tier or higher.
- A [supported data source](#).
- A [supported embedding model](#).
- Familiarity with the wizard. See [Import data wizards in the Azure portal](#).

Supported data sources

The wizard [supports a wide range of Azure data sources](#). However, this quickstart only covers the data sources that work with whole files, which are described in the following table.

 [Expand table](#)

Supported data source	Description
Azure Blob Storage	This data source works with blobs and tables. You must use a standard performance (general-purpose v2) account. Access tiers can be hot, cool, or cold.
Azure Data Lake Storage (ADLS) Gen2	This is an Azure Storage account with a hierarchical namespace enabled. To confirm that you have Data Lake Storage, check the Properties tab on the Overview page.

Supported data source	Description
	<p>Microsoft OneLake</p>

Supported embedding models

For integrated vectorization, use one of the following embedding models. Deployment instructions are provided in a [later section](#).

 Expand table

Provider	Supported models
Azure OpenAI in Azure AI Foundry Models resource ¹ ²	For text: text-embedding-ada-002 text-embedding-3-small text-embedding-3-large
Azure AI Foundry project	For text: text-embedding-ada-002 text-embedding-3-small text-embedding-3-large
Azure AI Foundry hub-based project	For text: text-embedding-ada-002 text-embedding-3-small text-embedding-3-large
Azure AI services multi-service resource ⁴	For text and images: Cohere-embed-v3-english ³ Cohere-embed-v3-multilingual ³
Azure AI services multi-service resource ⁴ ⁵	For text and images: Azure AI Vision multimodal

¹ The endpoint of your Azure OpenAI resource must have a [custom subdomain](#), such as `https://my-unique-name.openai.azure.com`. If you created your resource in the [Azure portal](#), this subdomain was automatically generated during resource setup.

² Azure OpenAI resources (with access to embedding models) that were created in the [Azure AI Foundry portal](#) aren't supported. You must create an Azure OpenAI resource in the Azure portal.

³ To use this model in the wizard, you must [deploy it as a serverless API deployment](#).

⁴ For billing purposes, you must [attach your Azure AI multi-service resource](#) to the skillset in your Azure AI Search service. Unless you use a [keyless connection \(preview\)](#) to create the skillset, both resources must be in the same region.

⁵ The Azure AI Vision multimodal embeddings APIs are available in [select regions](#).

Public endpoint requirements

For this quickstart, all of the preceding resources must have public access enabled so that the Azure portal nodes can access them. Otherwise, the wizard fails. After the wizard runs, you can enable firewalls and private endpoints on the integration components for security. For more information, see [Secure connections in the import wizards](#).

If private endpoints are already present and you can't disable them, the alternative option is to run the respective end-to-end flow from a script or program on a virtual machine. The virtual machine must be on the same virtual network as the private endpoint. Here's a [Python code sample](#) for integrated vectorization. The same [GitHub repo](#) has samples in other programming languages.

Role-based access

You can use Microsoft Entra ID with role assignments or key-based authentication with full-access connection strings. For Azure AI Search connections to other resources, we recommend role assignments. This quickstart assumes roles.

Free search services support role-based connections to Azure AI Search. However, they don't support managed identities on outbound connections to Azure Storage or Azure AI Vision. This lack of support requires key-based authentication on connections between free search services and other Azure resources. For more secure connections, use the Basic tier or higher, and then enable roles and configure a managed identity.

To configure the recommended role-based access:

1. On your search service, [enable roles](#) and [configure a system-assigned managed identity](#).
2. [Assign the following roles](#) to yourself.
 - **Search Service Contributor**
 - **Search Index Data Contributor**
 - **Search Index Data Reader**
3. On your data source and embedding model provider, create role assignments that allow your search service to access data and models. See [Prepare sample data](#) and [Prepare embedding models](#).

 **Note**

If you can't progress through the wizard because options aren't available (for example, you can't select a data source or an embedding model), revisit the role assignments. Error messages indicate that models or deployments don't exist, when the real cause is that the search service doesn't have permission to access them.

Check for space

If you're starting with the free service, you're limited to three indexes, data sources, skillsets, and indexers. Basic limits you to 15. This quickstart creates one of each object, so make sure you have room for extra items before you begin.

Prepare sample data

In this section, you use a [supported data source](#) to prepare sample data. Before you proceed, make sure you completed the prerequisites for [role-based access](#).

Azure Blob Storage

1. Sign in to the [Azure portal](#) and select your Azure Storage account.
2. From the left pane, select **Data storage > Containers**.
3. Create a container, and then upload the [health-plan PDF documents](#) used for this quickstart.
4. To assign roles:

- a. From the left pane, select **Access Control (IAM)**.
 - b. Select **Add > Add role assignment**.
 - c. Under **Job function roles**, select **Storage Blob Data Reader**, and then select **Next**.
 - d. Under **Members**, select **Managed identity**, and then select **Select members**.
 - e. Select your subscription and the managed identity of your search service.
5. (Optional) Synchronize deletions in your container with deletions in the search index.
To configure your indexer for deletion detection:
- a. [Enable soft delete](#) on your storage account. If you're using [native soft delete](#), the next step isn't required.
 - b. [Add custom metadata](#) that an indexer can scan to determine which blobs are marked for deletion. Give your custom property a descriptive name. For example, you can name the property "IsDeleted" and set it to false. Repeat this step for every blob in the container. When you want to delete the blob, change the property to true. For more information, see [Change and delete detection when indexing from Azure Storage](#).

Prepare embedding model

In this section, you deploy a [supported embedding model](#) for later use in this quickstart. Before you proceed, make sure you completed the prerequisites for [role-based access](#).

Azure OpenAI

The wizard supports several embedding models. Internally, the wizard calls the [Azure OpenAI Embedding skill](#) to connect to Azure OpenAI.

1. To assign roles:
 - a. Sign in to the [Azure portal](#) and select your Azure OpenAI resource.
 - b. From the left pane, select **Access control (IAM)**.
 - c. Select **Add > Add role assignment**.
 - d. Under **Job function roles**, select **Cognitive Services OpenAI User**, and then select **Next**.

- e. Under **Members**, select **Managed identity**, and then select **Select members**.
 - f. Select your subscription and the managed identity of your search service.
2. To deploy an embedding model:
- a. Sign in to the [Azure AI Foundry portal](#) and select your Azure OpenAI resource.
 - b. From the left pane, select **Model catalog**.
 - c. Deploy a [supported embedding model](#).

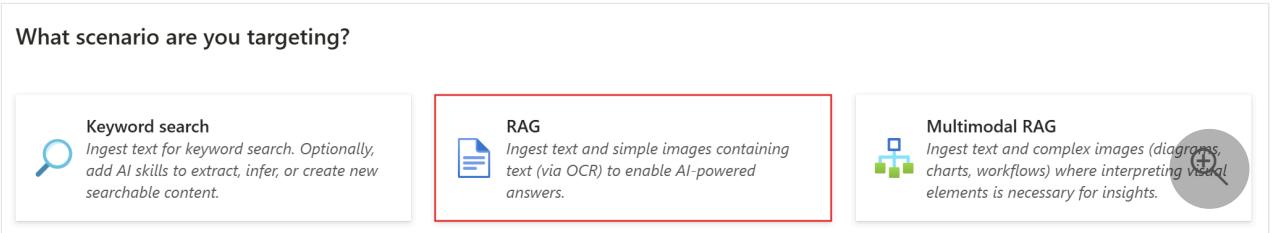
Start the wizard

To start the wizard for vector search:

1. Sign in to the [Azure portal](#) and select your Azure AI Search service.
2. On the **Overview** page, select **Import data (new)**.



3. Select your data source: **Azure Blob Storage**, **ADLS Gen2**, or **OneLake**.
4. Select **RAG**.



Connect to your data

In this step, you connect Azure AI Search to your chosen [data source](#) for content ingestion and indexing.

Azure Blob Storage

1. On the **Connect to your data** page, select your Azure subscription.
2. Select the storage account and container that provide the sample data.

3. If you enabled soft delete and added custom metadata in [Prepare sample data](#), select the **Enable deletion tracking** checkbox.

- On subsequent indexing runs, the search index is updated to remove any search documents based on soft-deleted blobs on Azure Storage.
- Blobs support either **Native blob soft delete** or **Soft delete using custom metadata**.
- If you configured your blobs for soft delete, provide the metadata property name-value pair. We recommend **IsDeleted**. If **IsDeleted** is set to **true** on a blob, the indexer drops the corresponding search document on the next indexer run.
- The wizard doesn't check Azure Storage for valid settings or throw an error if the requirements aren't met. Instead, deletion detection doesn't work, and your search index is likely to collect orphaned documents over time.

4. Select the **Authenticate using managed identity** checkbox. Leave the identity type as **System-assigned**.

Configure your Azure Blob Storage

Connect to your Azure Blob Storage containing PDFs and other unstructured data files. [Learn more](#)

Subscription *	My Azure Subscription
Storage account *	My Azure Storage Account
Blob container * ⓘ	health-plan-pdfs
Blob folder ⓘ	your/folder/here
<input checked="" type="checkbox"/> Enable deletion tracking ⓘ	
<input type="radio"/> Native blob soft delete	
<input checked="" type="radio"/> Soft delete using custom metadata	
Soft delete column *	IsDeleted
Delete marker value *	true
<input checked="" type="checkbox"/> Authenticate using managed identity. Learn more	
Managed identity type ⓘ	System-assigned

5. Select **Next**.

Vectorize your text

During this step, the wizard uses your chosen [embedding model](#) to vectorize chunked data. Chunking is built in and nonconfigurable. The effective settings are:

JSON

```
"textSplitMode": "pages",
"maximumPageLength": 2000,
"pageOverlapLength": 500,
"maximumPagesToTake": 0, #unlimited
"unit": "characters"
```

Azure OpenAI

1. On the **Vectorize your text** page, select **Azure OpenAI** for the kind.
2. Select your Azure subscription.
3. Select your Azure OpenAI resource, and then select the model you deployed in [Prepare embedding model](#).
4. For the authentication type, select **System assigned identity**.
5. Select the checkbox that acknowledges the billing effects of using these resources.

Vectorize your text

Connect to an Azure OpenAI, AI Studio or an Azure AI service and select an embedding model or multi-service account for vector generation. [Learn more](#)

Kind	<input type="text" value="Azure OpenAI"/>
Subscription *	<input type="text" value="Contoso Developer Division"/>
Azure OpenAI service * ⓘ	<input type="text" value="contosoazureopenaieastus"/>  Create a new Azure OpenAI service
Model deployment * ⓘ	<input type="text" value="text-embedding-3-large"/>
Authentication type ⓘ	<input type="radio"/> API key <input checked="" type="radio"/> System assigned identity <input type="radio"/> User assigned identity
<input checked="" type="checkbox"/> I acknowledge that connecting to an Azure OpenAI service will incur additional costs to my account. View pricing 	

6. Select **Next**.

Vectorize and enrich your images

The health-plan PDFs include a corporate logo, but otherwise, there are no images. You can skip this step if you're using the sample documents.

However, if your content includes useful images, you can apply AI in one or both of the following ways:

- Use a supported image embedding model from the Azure AI Foundry model catalog or the Azure AI Vision multimodal embeddings API (via an Azure AI multi-service resource) to vectorize images.
- Use optical character recognition (OCR) to extract text from images. This option invokes the [OCR skill](#).

Vectorize images

1. On the **Vectorize and enrich your images** page, select the **Vectorize images** checkbox.
2. For the kind, select your model provider: **AI Foundry Hub catalog models** or **AI Vision vectorization**.
If Azure AI Vision is unavailable, make sure your search service and multi-service resource are both in a [region that supports the Azure AI Vision multimodal APIs](#).
3. Select your Azure subscription, resource, and embedding model deployment (if applicable).
4. For the authentication type, select **System assigned identity** if you're not using a hub-based project. Otherwise, leave it as **API key**.
5. Select the checkbox that acknowledges the billing effects of using these resources.

Vectorize your images
Specify an embedding model that can create embeddings for image content.

Vectorize images ⓘ

Kind: AI Vision vectorization (Preview)

Subscription *: my-subscription

Select a multi-service account *: my-ai-multi-service-account

Create new AI service ⓘ

Authentication type *: ⓘ API key System assigned identity

I acknowledge that including Azure AI services involves a Pay-as-you-go pricing model. [View pricing](#) ⓘ

6. Select **Next**.

Add semantic ranking

On the [Advanced settings](#) page, you can optionally add [semantic ranking](#) to rerank results at the end of query execution. Reranking promotes the most semantically relevant matches to the top.

Map new fields

Key points about this step:

- The index schema provides vector and nonvector fields for chunked data.
- You can add fields, but you can't delete or modify generated fields.
- Document parsing mode creates chunks (one search document per chunk).

On the [Advanced settings](#) page, you can optionally add new fields, assuming the data source provides metadata or fields that aren't picked up on the first pass. By default, the wizard generates the fields described in the following table.

[] [Expand table](#)

Field	Applies to	Description
chunk_id	Text and image vectors	Generated string field. Searchable, retrievable, and sortable. This is the document key for the index.
parent_id	Text vectors	Generated string field. Retrievable and filterable. Identifies the parent document from which the chunk originates.
chunk	Text and image vectors	String field. Human readable version of the data chunk. Searchable and retrievable, but not filterable, facetable, or sortable.
title	Text and image vectors	String field. Human readable document title or page title or page number. Searchable and retrievable, but not filterable, facetable, or sortable.
text_vector	Text vectors	Collection(Edm.single). Vector representation of the chunk. Searchable and retrievable, but not filterable, facetable, or sortable.

You can't modify the generated fields or their attributes, but you can add new fields if your data source provides them. For example, Azure Blob Storage provides a collection of metadata fields.

1. Select **Add field**.

2. Select a source field from the available fields, enter a field name for the index, and accept (or override) the default data type.

 **Note**

Metadata fields are searchable but not retrievable, filterable, facetable, or sortable.

3. If you want to restore the schema to its original version, select **Reset**.

Schedule indexing

On the **Advanced settings** page, you can also specify an optional [run schedule](#) for the indexer. After you choose an interval from the dropdown list, select **Next**.

Finish the wizard

1. On the **Review your configuration** page, specify a prefix for the objects that the wizard creates. A common prefix helps you stay organized.
2. Select **Create**.

When the wizard completes the configuration, it creates the following objects:

- A data source connection.
- An index with vector fields, vectorizers, vector profiles, and vector algorithms. You can't design or modify the default index during the wizard workflow. Indexes conform to the [2024-05-01-preview REST API](#).
- A skillset with the [Text Split skill](#) for chunking and an embedding skill for vectorization. The embedding skill is either the [Azure OpenAI Embedding skill](#), [AML skill](#), or [Azure AI Vision multimodal embeddings skill](#). The skillset also has the [index projections](#) configuration, which maps data from one document in the data source to its corresponding chunks in a "child" index.
- An indexer with field mappings and output field mappings (if applicable).

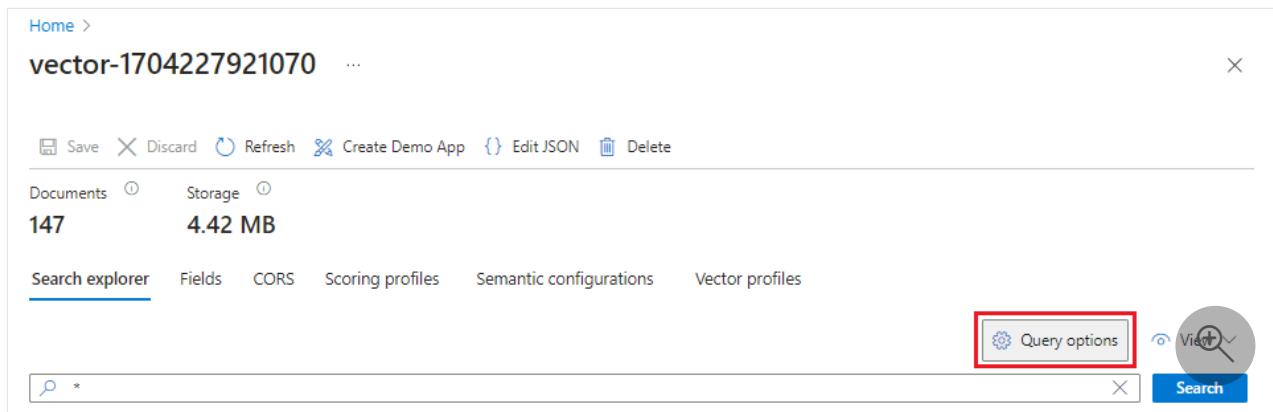
 **Tip**

Wizard-created objects have configurable JSON definitions. To view or modify these definitions, select **Search management** from the left pane, where you can view your indexes, indexers, data sources, and skillsets.

Check results

Search Explorer accepts text strings as input and then vectorizes the text for vector query execution.

1. In the Azure portal, go to **Search Management > Indexes**, and then select your index.
2. Select **Query options**, and then select **Hide vector values in search results**. This step makes the results more readable.



3. From the **View** menu, select **JSON view** so you can enter text for your vector query in the `text` vector query parameter.

vector-1701309912002

...

Save Discard Refresh Create Demo App Edit JSON Delete

Documents Storage 0 0 Bytes

Search explorer Fields CORS Scoring profiles Vector profiles

2023-10-01-Preview View

JSON query editor

```

1  {
2    "search": "*",
3    "select": "chunk, title",
4    "vectorQueries": [
5      {
6        "kind": "text",
7        "text": "Which plan has the lowest deductible?",  
8        "k": 5,
9        "fields": "vector"
10   ]

```

Query view

JSON view

Results

```

1  {
2    "@odata.context": "https://free-demo-search-svc.search.windows.net/indexes('vector-1701309912002')/$metadata#items",
3    "value": [
4      {
5        "@search.score": 0.835181,
6        "chunk": "year deductible is the same for \n\nall members of the plan and is reset each year on the plan",
7        "title": "Northwind_Health_Plus_Benefits_Details.pdf"
8      },

```

Search

The default query is an empty search ("*") but includes parameters for returning the number matches. It's a hybrid query that runs text and vector queries in parallel. It also includes semantic ranking and specifies which fields to return in the results through the `select` statement.

JSON

```

{
  "search": "*",
  "count": true,
  "vectorQueries": [
    {
      "kind": "text",
      "text": "*",
      "fields": "text_vector,image_vector"
    }
  ],
  "queryType": "semantic",
  "semanticConfiguration": "my-demo-semantic-configuration",
  "captions": "extractive",
  "answers": "extractive|count-3",
  "queryLanguage": "en-us",
  "select": "chunk_id,text_parent_id,chunk,title,image_parent_id"
}

```

- Replace both asterisk (*) placeholders with a question related to health plans, such as `Which plan has the lowest deductible?`.

JSON

```
{  
  "search": "Which plan has the lowest deductible?",  
  "count": true,  
  "vectorQueries": [  
    {  
      "kind": "text",  
      "text": "Which plan has the lowest deductible?",  
      "fields": "text_vector,image_vector"  
    }  
  ],  
  "queryType": "semantic",  
  "semanticConfiguration": "my-demo-semantic-configuration",  
  "captions": "extractive",  
  "answers": "extractive|count-3",  
  "queryLanguage": "en-us",  
  "select": "chunk_id,text_parent_id,chunk,title"  
}
```

5. To run the query, select Search.

The screenshot shows a search interface with a search bar containing the query "Which plan has the lowest deductible?". Below the search bar is a "Results" section. The results are presented as a list of JSON documents, each representing a chunk from a PDF. The first document in the list has the following structure:

```
33  "value": [  
34    {  
35      "@search.score": 0.017200259491801262,  
36      "@search.rerankerScore": 2.550845146179199,  
37      "@search.captions": [  
38        {  
39          "text": "the plan will pay towards the cost of care. The Allowed Amount may vary depending on the type of service.",  
40          "highlights": ""  
41        }  
42      ],  
43      "chunk_id": "62e612d57ba3_aHR0cHM6Ly9oZWIkaXN0c3RvcFmFnZWR1bW91YXN0dXMuYmxvYi5jb3JlLndpbmRvd3MubmV0L2h1YWxG",  
44      "text_parent_id": "aHR0cHM6Ly9oZWIkaXN0c3RvcFmFnZWR1bW91YXN0dXMuYmxvYi5jb3JlLndpbmRvd3MubmV0L2h1YWx0aC1wbG",  
45      "chunk": "the plan will pay towards the \ncost of care. The Allowed Amount may vary depending on the type",  
46      "title": "Northwind_Health_Plus_Benefits_Details.pdf",  
47      "image_parent_id": null  
48    },  
49    {  
50      "@search.score": 0.022132549434900284,  
51      "@search.rerankerScore": 2.492180347442627,  
52      "@search.captions": [  
53        {  
54          "text": "care physicians, specialists, hospitals, and pharmacies. This allows you to choose a provider",  
55          "highlights": ""  
56        }  
57      ],  
58      "chunk_id": "62e612d57ba3_aHR0cHM6Ly9oZWIkaXN0c3RvcFmFnZWR1bW91YXN0dXMuYmxvYi5jb3JlLndpbmRvd3MubmV0L2h1YWxG",  
59      "text_parent_id": "aHR0cHM6Ly9oZWIkaXN0c3RvcFmFnZWR1bW91YXN0dXMuYmxvYi5jb3JlLndpbmRvd3MubmV0L2h1YWx0aC1wbG",  
60      "chunk": "care physicians, specialists, hospitals, and pharmacies. This allows you to choose a provider",  
61      "title": "Northwind_Standard_Benefits_Details.pdf",  
62      "image_parent_id": null
```

Each document is a chunk of the original PDF. The `title` field shows which PDF the chunk comes from. Each `chunk` is long. You can copy and paste one into a text editor to read the entire value.

6. To see all of the chunks from a specific document, add a filter for the `title_parent_id` field for a specific PDF. You can check the **Fields** tab of your index to confirm the field is filterable.

JSON

```
{  
    "select": "chunk_id, text_parent_id, chunk, title",  
    "filter": "text_parent_id eq  
'aHR0cHM6Ly9oZWlkaXN0c3RvcnFnZWR1bW91YXN0dXMuYmxvYi5jb3JlLndpbmRvd3MubmV0L2h1YW  
x0aC1wbGFuLXBkZnMvTm9ydGh3aW5kX1N0YW5kYXJkX0J1bmVmaXRzX0R1dGFpbHMucGRm0'",  
    "count": true,  
    "vectorQueries": [  
        {  
            "kind": "text",  
            "text": "*",  
            "k": 5,  
            "fields": "text_vector"  
        }  
    ]  
}
```

Clean up resources

This quickstart uses billable Azure resources. If you no longer need the resources, delete them from your subscription to avoid charges.

Next step

This quickstart introduced you to the **Import data (new)** wizard, which creates all of the necessary objects for integrated vectorization. To explore each step in detail, see [Set up integrated vectorization in Azure AI Search](#).

Last updated on 10/24/2025

Quickstart: Search for multimodal content in the Azure portal

In this quickstart, you use the [Import data \(new\)](#) wizard in the Azure portal to get started with [multimodal search](#). The wizard simplifies the process of extracting, chunking, vectorizing, and loading both text and images into a searchable index.

Unlike [Quickstart: Vector search in the Azure portal](#), which processes simple text-containing images, this quickstart supports advanced image processing for multimodal RAG scenarios.

This quickstart uses a multimodal PDF from the [azure-search-sample-data](#) repo. However, you can use different files and still complete this quickstart.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#). We recommend the Basic tier or higher.
- An [Azure Storage account](#). Use Azure Blob Storage or Azure Data Lake Storage Gen2 (storage account with a hierarchical namespace) on a standard performance (general-purpose v2) account. Access tiers can be hot, cool, or cold.
- A [supported extraction method](#).
- A [supported embedding method](#).
- Familiarity with the wizard. See [Import data wizards in the Azure portal](#).

Supported extraction methods

For content extraction, you can choose either default extraction via Azure AI Search or enhanced extraction via [Azure Document Intelligence in Foundry Tools](#). The following table describes both extraction methods.

[] [Expand table](#)

Method	Description
Default extraction	Extracts location metadata from PDF images only. Doesn't require another Azure AI resource.
Enhanced	Extracts location metadata from text and images for multiple document types. Requires

Method	Description
extraction	a Microsoft Foundry resource ¹ in a supported region .

¹ For billing purposes, you must [attach your Foundry resource](#) to the skillset in your Azure AI Search service. Unless you use a [keyless connection](#) to create the skillset, both resources must be in the same region.

Supported embedding methods

For content embedding, choose one of the following methods:

- **Image verbalization:** Uses an LLM to generate natural-language descriptions of images, and then uses an embedding model to vectorize plain text and verbalized images.
- **Multimodal embeddings:** Uses an embedding model to directly vectorize both text and images.

The following table lists the supported providers and models for each method. Deployment instructions for the models are provided in a [later section](#).

[] [Expand table](#)

Provider	Models for image verbalization	Models for multimodal embeddings
Azure OpenAI in Foundry Models resource ^{1, 2}	LLMs: gpt-4o gpt-4o-mini gpt-5 gpt-5-mini gpt-5-nano Embedding models: text-embedding-ada-002 text-embedding-3-small text-embedding-3-large	
Foundry project	LLMs: phi-4 gpt-4o gpt-4o-mini gpt-5 gpt-5-mini gpt-5-nano Embedding models:	

Provider	Models for image verbalization	Models for multimodal embeddings
	text-embedding-ada-002 text-embedding-3-small text-embedding-3-large	
Foundry hub-based project	LLMs: phi-4 gpt-4o gpt-4o-mini gpt-5 gpt-5-mini gpt-5-nano Embedding models: text-embedding-ada-002 text-embedding-3-small text-embedding-3-large Cohere-embed-v3-english ³ Cohere-embed-v3-multilingual ³	Cohere-embed-v3-english ³ Cohere-embed-v3-multilingual ³
Foundry resource ⁴	Embedding model: Azure Vision in Foundry Tools multimodal ⁵	Azure Vision multimodal ⁵

¹ The endpoint of your Azure OpenAI resource must have a [custom subdomain](#), such as <https://my-unique-name.openai.azure.com>. If you created your resource in the [Azure portal](#) , this subdomain was automatically generated during resource setup.

² Azure OpenAI resources (with access to embedding models) that were created in the [Foundry portal](#)  aren't supported. You must create an Azure OpenAI resource in the Azure portal.

³ To use this model in the wizard, you must [deploy it as a serverless API deployment](#).

⁴ For billing purposes, you must [attach your Foundry resource](#) to the skillset in your Azure AI Search service. Unless you use a [keyless connection \(preview\)](#) to create the skillset, both resources must be in the same region.

⁵ The Azure Vision multimodal embeddings APIs are available in [select regions](#).

Public endpoint requirements

All of the preceding resources must have public access enabled so that the Azure portal nodes can access them. Otherwise, the wizard fails. After the wizard runs, you can enable firewalls and private endpoints on the integration components for security. For more information, see [Secure connections in the import wizards](#).

If private endpoints are already present and you can't disable them, the alternative is to run the respective end-to-end flow from a script or program on a virtual machine. The virtual machine must be on the same virtual network as the private endpoint. [Here's a Python code sample](#) for integrated vectorization. The same [GitHub repo](#) has samples in other programming languages.

Check for space

If you're starting with the free service, you're limited to three indexes, three data sources, three skillsets, and three indexers. Make sure you have room for extra items before you begin. This quickstart creates one of each object.

Configure access

Before you begin, make sure you have permissions to access content and operations. We recommend Microsoft Entra ID authentication and role-based access for authorization. You must be an **Owner** or **User Access Administrator** to assign roles. If roles aren't feasible, you can use [key-based authentication](#) instead.

Configure the [required roles](#) and [conditional roles](#) identified in this section.

Required roles

Azure AI Search and Azure Storage are required for all multimodal search scenarios.

Azure AI Search

Azure AI Search provides the multimodal pipeline. Configure access for yourself and your search service to read data, run the pipeline, and interact with other Azure resources.

On your Azure AI Search service:

1. [Enable role-based access](#).
2. [Configure a system-assigned managed identity](#).
3. [Assign the following roles](#) to yourself.
 - **Search Service Contributor**
 - **Search Index Data Contributor**

- Search Index Data Reader

Conditional roles

The following tabs cover all wizard-compatible resources for multimodal search. Select only the tabs that apply to your chosen [extraction method](#) and [embedding method](#).

Azure OpenAI

Azure OpenAI provides LLMs for image verbalization and embedding models for text and image vectorization. Your search service requires access to call the [GenAI Prompt skill](#) and [Azure OpenAI Embedding skill](#).

On your Azure OpenAI resource:

- Assign [Cognitive Services OpenAI User](#) to your [search service identity](#).

Prepare sample data

This quickstart uses a sample multimodal PDF, but you can also use your own files. If you're on a free search service, use fewer than 20 files to stay within the free quota for enrichment processing.

To prepare the sample data for this quickstart:

1. Sign in to the [Azure portal](#) and select your Azure Storage account.
2. From the left pane, select **Data storage > Containers**.
3. Create a container, and then upload the [sample PDF](#) to the container.
4. Create another container to store images extracted from the PDF.

Deploy models

The wizard offers several options for content embedding. Image verbalization requires an LLM to describe images and an embedding model to vectorize text and image content, while direct multimodal embeddings only require an embedding model. These models are available through Azure OpenAI and Foundry.

(!) Note

If you're using Azure Vision, skip this step. The multimodal embeddings are built into your Foundry resource and don't require model deployment.

To deploy the models for this quickstart:

1. Sign in to the [Foundry portal](#).
2. Select your Azure OpenAI resource or Foundry project.
3. Deploy the models required for your chosen [embedding method](#).

Start the wizard

To start the wizard for multimodal search:

1. Sign in to the [Azure portal](#) and select your Azure AI Search service.
2. On the **Overview** page, select **Import data (new)**.



3. Select your data source: **Azure Blob Storage** or **Azure Data Lake Storage Gen2**.

A screenshot of the 'Import data (new)' wizard. The top bar says 'Let's start by picking a data source...'. Below it is a 'Filter by name...' input field. A list of data sources is shown in a grid:

- Azure Blob Storage** (Built-in indexer)
- Azure Data Lake Storage Gen2** (Built-in indexer)
- Azure Cosmos DB** (Built-in indexer)
- Azure SQL Database** (Built-in indexer)
- Azure Table Storage** (Built-in indexer)
- Fabric OneLake files (Preview)** (Built-in indexer)
- SharePoint** (Logic Apps indexer)
- OneDrive** (Logic Apps indexer)
- OneDrive for Business** (Logic Apps indexer)
- Azure File Storage** (Logic Apps indexer)
- Azure Queues** (Logic Apps indexer)
- Service Bus** (Logic Apps indexer)
- Amazon S3** (Logic Apps indexer)
- Dropbox** (Logic Apps indexer)
- SFTP - SSH** (Logic Apps indexer)

A magnifying glass icon is in the bottom right corner of the grid.

4. Select **Multimodal RAG**.





Connect to your data

Azure AI Search requires a connection to a data source for content ingestion and indexing. In this case, the data source is your Azure Storage account.

To connect to your data:

1. On the **Connect to your data** page, select your Azure subscription.
2. Select the storage account and container to which you uploaded the sample data.
3. Select the **Authenticate using managed identity** checkbox. Leave the identity type as **System-assigned**.

Configure your Azure Blob Storage

Connect to your Azure Blob Storage containing PDFs, other unstructured data files, and structured data files. [Learn more](#)

Subscription *	my-subscription
Storage account *	mystorageaccount
Blob container * ⓘ	my-container
Blob folder ⓘ	your/folder/here

Enable deletion tracking ⓘ

Authenticate using managed identity. [Learn more](#)

Managed identity type ⓘ System-assigned

4. Select **Next**.

Extract your content

Depending on your chosen [extraction method](#), the wizard provides configuration options for document cracking and chunking.

Default extraction

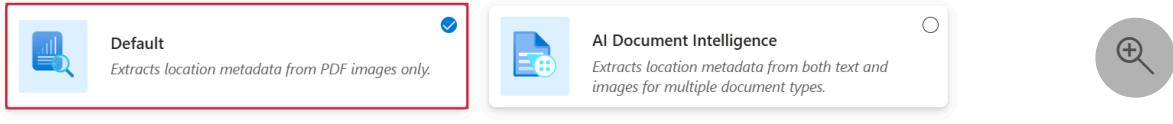
The default method calls the [Document Extraction skill](#) to extract text content and generate normalized images from your documents. The [Text Split skill](#) is then called to split the extracted text content into pages.

To use the Document Extraction skill:

1. On the Content extraction page, select Default.

Content extraction

Choose a content extraction path that suits your scenario. These paths extract location metadata such as page numbers, bounding polygons, annotations; and other metadata and content from text and images. Learn more about the pricing options for [default content extraction](#) and [AI Document Intelligence](#).



2. Select Next.

Embed your content

During this step, the wizard uses your chosen [embedding method](#) to generate vector representations of both text and images.

Image verbalization

The wizard calls one skill to create descriptive text for images (image verbalization) and another skill to create vector embeddings for both text and images.

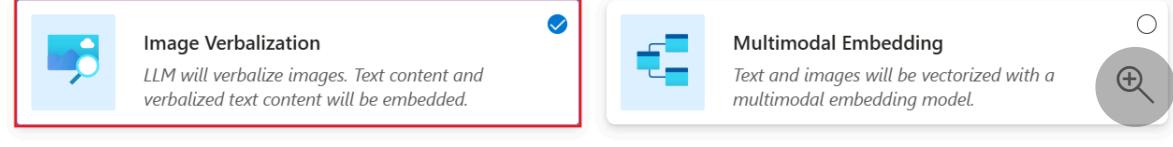
For image verbalization, the [GenAI Prompt skill](#) uses your deployed LLM to analyze each extracted image and produce a natural-language description.

For embeddings, the [Azure OpenAI Embedding skill](#), [AML skill](#), or [Azure Vision multimodal embeddings skill](#) uses your deployed embedding model to convert text chunks and verbalized descriptions into high-dimensional vectors. These vectors enable similarity and hybrid retrieval.

To use the skills for image verbalization:

1. On the Content embedding page, select Image Verbalization.

Choose your content embedding strategy



2. On the Image Verbalization tab:

- a. For the kind, select your LLM provider: **Azure OpenAI** or **Foundry Hub catalog models**.
- b. Select your Azure subscription, resource, and LLM deployment.
- c. For the authentication type, select **System assigned identity** if you're not using a hub-based project. Otherwise, leave it as **API key**.
- d. Select the checkbox that acknowledges the billing effects of using these resources.

① Image Verbalization ② Text Vectorization

(i) Please complete the text vectorization step to proceed.

Kind	Azure OpenAI
Subscription *	my-subscription
Azure OpenAI service * ⓘ	my-aoai-resource
Create a new Azure OpenAI service ↗	
Model deployment * ⓘ	gpt-4o
Authentication type ⓘ	<input type="radio"/> API key <input checked="" type="radio"/> System assigned identity <input type="radio"/> User assigned identity
<input checked="" type="checkbox"/> I acknowledge that connecting to an Azure OpenAI service will incur additional costs to my account. View pricing ↗	

3. On the **Text Vectorization** tab:

- a. For the kind, select your model provider: **Azure OpenAI**, **Foundry Hub catalog models**, or **AI Vision vectorization**.
- b. Select your Azure subscription, resource, and embedding model deployment (if applicable).
- c. For the authentication type, select **System assigned identity** if you're not using a hub-based project. Otherwise, leave it as **API key**.
- d. Select the checkbox that acknowledges the billing effects of using these resources.

1 Image Verbalization **2 Text Vectorization**

Kind	Azure OpenAI
Subscription *	my-subscription
Azure OpenAI service * ⓘ	my-aoai-resource
Create a new Azure OpenAI service	
Model deployment * ⓘ	text-embedding-3-large
Authentication type ⓘ	<input type="radio"/> API key <input checked="" type="radio"/> System assigned identity <input type="radio"/> User assigned identity
<input checked="" type="checkbox"/> I acknowledge that connecting to an Azure OpenAI service will incur additional costs to my account. View pricing	



4. Select Next.

Store the extracted images

The next step is to send images extracted from your documents to Azure Storage. In Azure AI Search, this secondary storage is known as a [knowledge store](#).

To store the extracted images:

1. On the **Image output** page, select your Azure subscription.
2. Select the storage account and blob container you created to store the images.
3. Select the **Authenticate using managed identity** checkbox. Leave the identity type as **System-assigned**.

Image output location

Select a storage account to store the images extracted from your documents.

Subscription *	my-subscription
Storage account *	mystorageaccount
Blob container * ⓘ	my-output-container
<input checked="" type="checkbox"/> Authenticate using managed identity. Learn more	
Managed identity type ⓘ	System-assigned



4. Select Next.

Map new fields

On the **Advanced settings** page, you can optionally add fields to the index schema. By default, the wizard generates the fields described in the following table.

[+] Expand table

Field	Applies to	Description	Attributes
content_id	Text and image vectors	String field. Document key for the index.	Retrievable, sortable, and searchable.
document_title	Text and image vectors	String field. Human-readable document title.	Retrievable and searchable.
text_document_id	Text vectors	String field. Identifies the parent document from which the text chunk originates.	Retrievable and filterable.
image_document_id	Image vectors	String field. Identifies the parent document from which the image originates.	Retrievable and filterable.
content_text	Text vectors	String field. Human-readable version of the text chunk.	Retrievable and searchable.
content_embedding	Text and image vectors	Collection(Edm.Single). Vector representation of text and images.	Retrievable and searchable.
content_path	Text and image vectors	String field. Path to the content in the storage container.	Retrievable and searchable.
locationMetadata	Image vectors	Edm.ComplexType. Contains metadata about the image's location in the documents.	Varies by field.

You can't modify the generated fields or their attributes, but you can add fields if your data source provides them. For example, Azure Blob Storage provides a collection of metadata fields.

To add fields to the index schema:

1. Under **Index fields**, select **Preview and edit**.
2. Select **Add field**.
3. Select a source field from the available fields, enter a field name for the index, and accept (or override) the default data type.

4. If you want to restore the schema to its original version, select **Reset**.

Schedule indexing

For data sources where the underlying data is volatile, you can [schedule indexing](#) to capture changes at specific intervals or specific dates and times.

To schedule indexing:

1. On the **Advanced settings** page, under **Schedule indexing**, specify a run schedule for the indexer. We recommend **Once** for this quickstart.



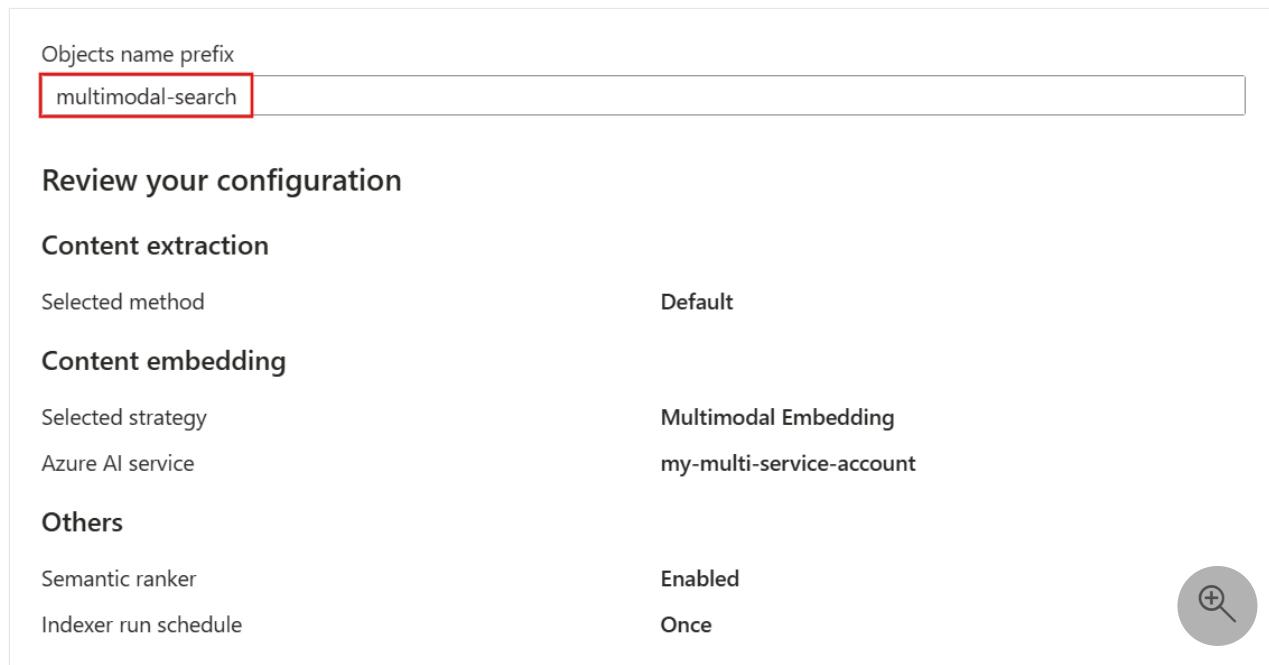
2. Select **Next**.

Finish the wizard

The final step is to review your configuration and create the necessary objects for multimodal search. If necessary, return to the previous pages in the wizard to adjust your configuration.

To finish the wizard:

1. On the **Review and create** page, specify a prefix for the objects the wizard will create. A common prefix helps you stay organized.



2. Select **Create**.

When the wizard completes the configuration, it creates the following objects:

- An indexer that drives the indexing pipeline.
- A data source connection to Azure Blob Storage.
- An index with text fields, vector fields, vectorizers, vector profiles, and vector algorithms. During the wizard workflow, you can't modify the default index. Indexes conform to the [2024-05-01-preview REST API](#) so that you can use preview features.
- A skillset with the following skills:
 - The [Document Extraction skill](#) or [Document Layout skill](#) extracts text and images from source documents. The [Text Split skill](#) accompanies the Document Extraction skill for data chunking, while the Document Layout skill has built-in chunking.
 - The [GenAI Prompt skill](#) verbalizes images in natural language. If you're using direct multimodal embeddings, this skill is absent.
 - The [Azure OpenAI Embedding skill](#), [AML skill](#), or [Azure Vision multimodal embeddings skill](#) is called once for text vectorization and once for image vectorization.
 - The [Shaper skill](#) enriches the output with metadata and creates new images with contextual information.

Tip

Wizard-created objects have configurable JSON definitions. To view or modify these definitions, select **Search management** from the left pane, where you can view your indexes, indexers, data sources, and skillsets.

Check results

This quickstart creates a multimodal index that supports [hybrid search](#) over both text and images. Unless you use direct multimodal embeddings, the index doesn't accept images as query inputs, which requires the [AML skill](#) or [Azure Vision multimodal embeddings skill](#) with an equivalent vectorizer. For more information, see [Configure a vectorizer in a search index](#).

Hybrid search combines full-text queries and vector queries. When you issue a hybrid query, the search engine computes the semantic similarity between your query and the indexed

vectors and ranks the results accordingly. For the index created in this quickstart, the results surface content from the `content_text` field that closely aligns with your query.

To query your multimodal index:

1. Sign in to the [Azure portal](#) and select your Azure AI Search service.
2. From the left pane, select **Search management > Indexes**.
3. Select your index.
4. Select **Query options**, and then select **Hide vector values in search results**. This step makes the results more readable.

The screenshot shows the Azure AI Search service interface for the 'multimodal-index'. The 'Search explorer' tab is active. At the top, there are buttons for Save, Discard, Refresh, Create demo app, Edit JSON, Delete, and Encryption. Below that, it shows document statistics: 105 documents, 4.5 MB total storage, 1.24 MB vector index quota usage, and 15 GB max storage. There are also tabs for Fields, CORS, Scoring profiles, Semantic configurations, and Vector profiles. On the right side, there are 'Query options' and 'View' buttons, with 'Query options' being highlighted by a red box. Below the tabs is a search bar containing 'energy'.

5. Enter text for which you want to search. Our example uses `energy`.

6. To run the query, select **Search**.

The screenshot shows the 'Search explorer' interface. The search bar at the bottom contains the text 'energy'. To the right of the search bar are 'Query options' and 'View' buttons, with 'View' having a dropdown arrow. Below the search bar is a 'Search' button, which is highlighted with a red box.

The JSON results should include text and image content related to `energy` in your index. If you enabled semantic ranker, the `@search.answers` array provides concise, high-confidence [semantic answers](#) to help you quickly identify relevant matches.

The screenshot shows the JSON results for the search query 'energy'. The results are displayed in a JSON editor. The first result under the `@search.answers` array is shown in red. It includes a key and a value. The value is a long string of alphanumeric characters followed by '_normalized_images_7'. Below this, there is another object with a 'text' field containing a detailed description of AI's role in sustainability.

```
"@search.answers": [
  {
    "key": "a71518188062_aHR0cHM6Ly9oYWlsZX1zdG9yYWd1LmJsb2IuY29yZS53aw5kb3dzLm51dC9tdWx0aW1vZGFsLXNlYXJjaC9BY2NlbGVyYXRpbmcU3VzdGFpbmFiaWxpdHktd210aC1BSS0yMDI1LnBkZg2_normalized_images_7",
    "text": "A vertical infographic consisting of three sections describing the roles of AI in sustainability: 1. **Measure, predict, and optimize complex systems**: AI facilitates analysis, modeling, and optimization in areas like energy distribution, resource allocation, and environmental monitoring."
  }
]
```

```
    "highlight": "Accelerate the development of sustainability solution...",  
    "highlights": "A vertical infographic consisting of three sections  
describing the roles of AI in sustainability: 1. **Measure, predict, and  
optimize complex systems**: AI facilitates analysis, modeling, and optimization  
in areas like<em> energy distribution, </em>resource<em> allocation, </em>and  
environmental monitoring. **Accelerate the development of sustainability  
solution...”,  
    "score": 0.9950000047683716  
,  
{  
    "key":  
"1cb0754930b6_aHR0cHM6Ly9oYWlsZXlzdG9yYWdlLmJsb2IuY29yZS53aW5kb3dzLm5ldC9tdWx0a  
W1vZGFsLXN1YXJjaC9BY2NbGVyYXRpbmctU3VzdGFpbmFiaWxpdHktd2l0aC1BSS0yMDI1LnBkZg2_  
text_sections_5",  
    "text": "...cross-laminated timber.8 Through an agreement with  
Brookfield, we aim 10.5 gigawatts (GW) of renewable energy to the grid.910.5  
GWof new renewable energy capacity to be developed across the United States and  
Europe.Play 4 Advance AI policy principles and governance for sustainabilityWe  
advocated for policies that accelerate grid decarbonization",  
    "highlight": "...cross-laminated timber.8 Through an agreement with  
Brookfield, we aim <em> 10.5 gigawatts (GW) of renewable energy </em>to the<em>  
grid.910.5 </em>GWof new<em> renewable energy </em>capacity to be developed  
across the United States and Europe.Play 4 Advance AI policy principles and  
governance for sustainabilityWe advocated for policies that accelerate grid  
decarbonization",  
    "score": 0.9890000224113464  
,  
{  
    "key":  
"1cb0754930b6_aHR0cHM6Ly9oYWlsZXlzdG9yYWdlLmJsb2IuY29yZS53aW5kb3dzLm5ldC9tdWx0a  
W1vZGFsLXN1YXJjaC9BY2NbGVyYXRpbmctU3VzdGFpbmFiaWxpdHktd2l0aC1BSS0yMDI1LnBkZg2_  
text_sections_50",  
    "text": "ForewordAct... Similarly, we have restored degraded stream  
ecosystems near our datacenters from Racine, Wisconsin120 to Jakarta,  
Indonesia.117INNOVATION SPOTLIGHTAI-powered Community Solar  
MicrogridsDeveloping energy transition programsWe are co-innovating with  
communities to develop energy transition programs that align their goals with  
broader s.",  
    "highlight": "ForewordAct... Similarly, we have restored degraded stream  
ecosystems near our datacenters from Racine, Wisconsin120 to Jakarta,  
Indonesia.117INNOVATION SPOTLIGHTAI-powered Community<em> Solar  
MicrogridsDeveloping energy transition programsWe </em>are co-innovating with  
communities to develop<em> energy transition programs </em>that align their  
goals with broader s.",  
    "score": 0.9869999885559082  
}  
]
```

Clean up resources

This quickstart uses billable Azure resources. If you no longer need the resources, delete them from your subscription to avoid charges.

Next steps

This quickstart introduced you to the **Import data (new)** wizard, which creates all of the necessary objects for multimodal search. To explore each step in detail, see the following tutorials:

- [Tutorial: Verbalize images using generative AI](#)
 - [Tutorial: Verbalize images from a structured document layout](#)
 - [Tutorial: Vectorize images and text](#)
 - [Tutorial: Vectorize from a structured document layout](#)
-

Last updated on 11/18/2025

Quickstart: Create a demo search app in the Azure portal

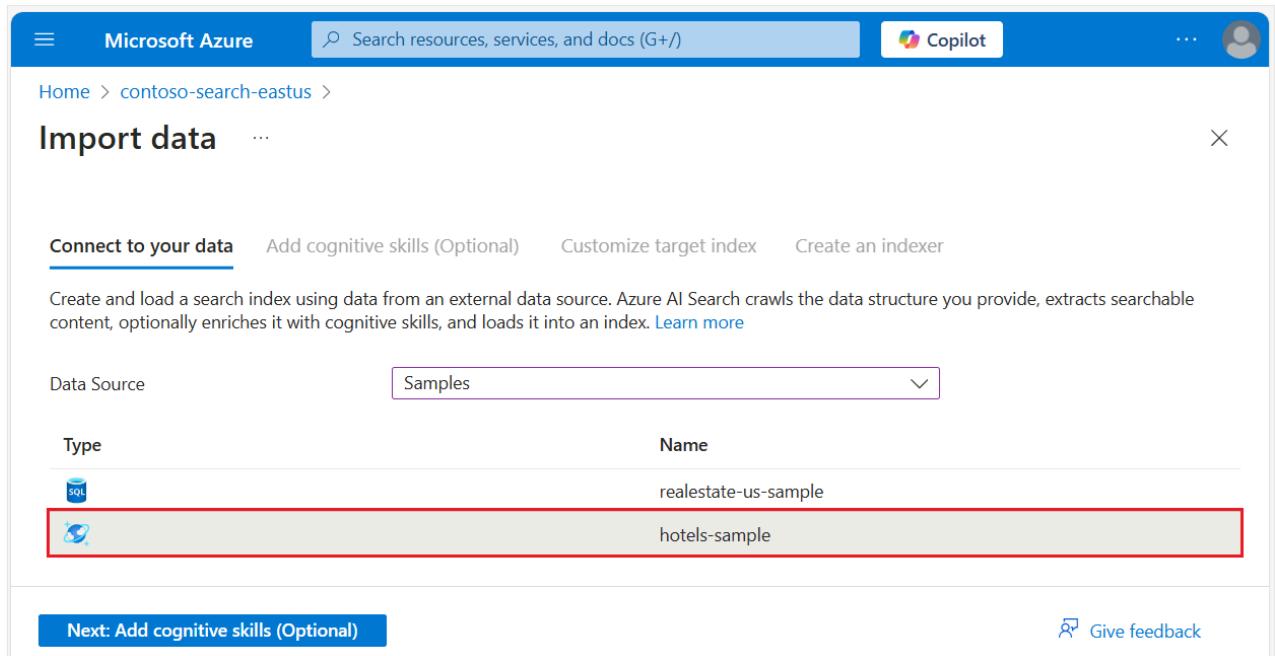
In this quickstart, you use the [Create demo app](#) wizard in the Azure portal to generate a downloadable, "localhost"-style web app that runs in a browser. Depending on how you configure it, the generated app is operational on first use, with a live read-only connection to an index on your search service. A default app can include a search box, results area, sidebar filters, and typeahead support.

A demo app can help you visualize how an index functions in a client app, but it isn't intended for production scenarios. Production apps should include security, error handling, and hosting logic that the demo app doesn't provide.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription. For this quickstart, you can use a free service.
- A [search index](#) to use as the basis of your generated application.

This quickstart uses the hotels-sample index. Follow the instructions in [this quickstart](#) to create the index.



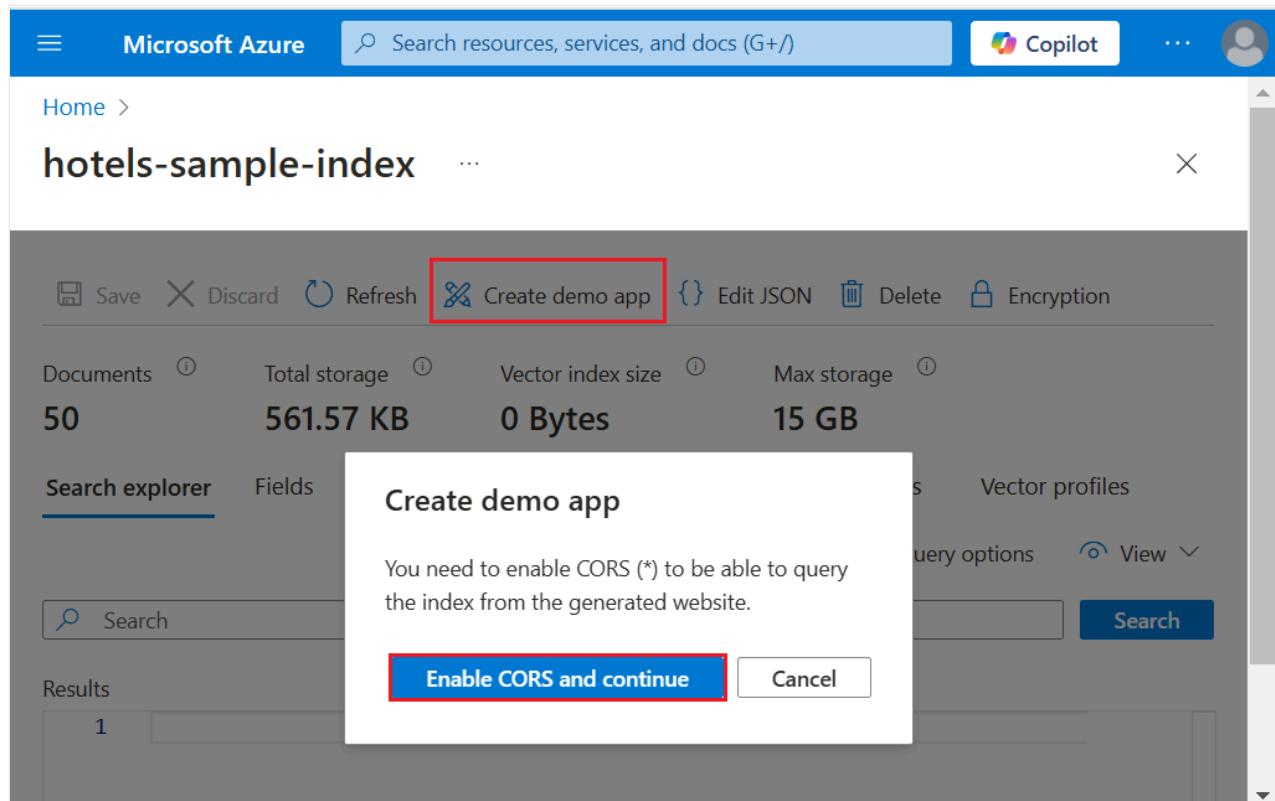
The screenshot shows the Microsoft Azure portal interface with the following details:

- Header:** Microsoft Azure, Search resources, services, and docs (G+), Copilot, and a user profile icon.
- Breadcrumb:** Home > contoso-search-eastus >
- Title:** Import data
- Subtitles:** Connect to your data, Add cognitive skills (Optional), Customize target index, Create an indexer.
- Description:** Create and load a search index using data from an external data source. Azure AI Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)
- Data Source:** Samples (selected)
- Type:** SQL (selected)
- Name:** realestate-us-sample
- Recent Data Sources:** hotels-sample (highlighted with a red border)
- Buttons:** Next: Add cognitive skills (Optional) and Give feedback.

Start the wizard

To start the wizard for this quickstart:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Search management > Indexes**.
3. Select **hotels-sample-index** from the list.
4. At the top of the index page, select **Create demo app**.
5. Select **Enable CORS** and continue to add CORS support to your index definition.



Configure search results

The wizard provides a basic layout for the rendered search results, including space for a thumbnail image, title, and description. Each element is backed by a field in your index that provides the necessary data.

To configure the search results:

1. Skip **Thumbnail** because the index doesn't have image URLs.

However, if your index contains a field populated with URLs that resolve to publicly available images, you should specify that field for the thumbnail.

2. For **Title**, choose a field that conveys the uniqueness of each document. Our example uses **HotelName**.

3. For **Description**, choose a field that might help someone decide whether to drill down to that particular document. Our example uses **Description**.

4. Select **Next**.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header bar with the Microsoft Azure logo, a search bar containing 'Search resources, services, and docs (G+)', and a 'Copilot' button. Below the header, the URL 'Home > contoso-search-eastus | Indexes > hotels-sample-index >' is visible. The main content area has a title 'Create demo app' with a close button. On the left, a vertical list of options is shown: 'Customize individual result' (selected, indicated by a blue dot), 'Customize sidebar', and 'Customize suggestions'. To the right of this list is a preview window showing a simple search result card with a placeholder 'Thumbnail' and dropdown menus for 'Title' (set to 'HotelName') and 'Description'. Below the preview are buttons for 'Previous' and 'Next', and a magnifying glass icon. A descriptive text block above the preview says: 'Customize how individual results will be displayed.' and 'An individual result: Choose the look and feel for each search result below. The layout corresponds to how each result will be rendered.'

Add a sidebar

The search service supports faceted navigation, which is often rendered as a sidebar. Facets are based on fields attributed as filterable and facetable in your index schema.

Tip

To view field attributes, select the **Fields** tab on the index page in the Azure portal. Only fields marked as filterable and facetable can be used in the sidebar.

In Azure AI Search, faceted navigation is a cumulative filtering experience. Within a category, selecting multiple filters expands the results, such as selecting both `Seattle` and `Bellevue` within the `city` filter. Across categories, selecting multiple filters narrows the results.

To customize the sidebar:

1. Review the list of filterable and facetable fields in the index.

2. To shorten the sidebar and prevent scrolling in the finished app, delete some fields.

3. Select **Next**.

The screenshot shows the Microsoft Azure portal interface for creating a demo app. The top navigation bar includes 'Microsoft Azure', a search bar, and a user profile icon. Below the navigation, the breadcrumb trail shows 'Home > contoso-search-eastus | Indexes > hotels-sample-index'. The main title is 'Create demo app' with a '...' button. On the left, there's a vertical navigation menu with three items: 'Customize individual result' (selected, indicated by a checked checkbox), 'Customize sidebar' (indicated by a blue circle), and 'Customize suggestions' (indicated by an empty circle). The main content area is titled 'UI components of sidebar' with the sub-instruction 'Choose which fields are displayed in the filter sidebar. Only fields that are "filterable" and "facetable" can be used.' It features a diagram of a sidebar with a list of items. Below this is a table where users can map fields to filtering components. The table has two columns: 'Field name' and 'Filtering component'. The fields listed are HotelId, Category, Tags, Rating, Address/City, Address/StateProvince, Address/PostalCode, Address/Country, and Rooms/Type. Each field is mapped to a 'Checkbox' component. There are also 'Add', 'Delete', and '...' buttons for each row. At the bottom of the table are 'Previous' and 'Next' buttons, and a magnifying glass icon.

Add suggestions

Suggestions are automated query prompts that appear in the search box. The demo app supports suggestions that provide a dropdown list of potential matching documents based on partial text inputs.

To customize the suggestions:

1. Choose the fields you want to display as suggested queries. Use shorter string fields instead of verbose fields, such as descriptions.
2. Use the **Show Field Name** checkbox to include or exclude labels for the suggestions.

Microsoft Azure Search resources, services, and docs (G+/)

Copilot

Home > contoso-search-eastus | Indexes > hotels-sample-index >

Create demo app

Customize individual result

Customize sidebar

Customize suggestions

Choose and customize which fields will be displayed as suggestion in search box dropdown.

Suggestions of search box

Choose the look and feel for each search suggestion below. The order of the table corresponds to the order that the fields are displayed for each result.

+ Add Delete

Style	Field name	Show Field Name
Normal	HostName	<input checked="" type="checkbox"/>
Normal	Category	<input type="checkbox"/>
Normal	Address.City	<input type="checkbox"/>

Previous Create demo app

The screenshot shows the 'Create demo app' wizard in the Microsoft Azure portal. On the left, there's a vertical list of customization steps: 'Customize individual result' (checked), 'Customize sidebar' (checked), and 'Customize suggestions' (unchecked). The 'Customize sidebar' section contains a placeholder window icon and a note about choosing fields for the search box dropdown. The 'Customize suggestions' section is titled 'Suggestions of search box' and includes a table for setting styles and field names. The table has three rows: 'Normal' style for 'HostName' with 'Show Field Name' checked; 'Normal' style for 'Category' with it unchecked; and 'Normal' style for 'Address.City' with it unchecked. At the bottom are 'Previous' and 'Create demo app' buttons, and a magnifying glass icon.

Create, download, and execute

To finish the wizard and use the demo app:

1. Select **Create demo app** to generate the HTML file.
2. When prompted, select **Download** to download the file.
3. Open the file in a browser.
4. Select the search button to run an empty query (*) that returns an arbitrary result set.
5. Enter a term in the search box and use the sidebar filters to narrow the results.

The screenshot shows a search interface for 'OCEAN WATER RESORT & SPA' and 'HOTEL ON THE HARBOR'. On the left, there are filter panels for 'Category' (Resort and Spa, Suite, Luxury, Boutique) and 'Tags' (pool, bar, continental breakfast, free wifi, air conditioning, free parking, 24-hour front desk service). A search bar at the top right contains a magnifying glass icon. A sidebar on the right includes a 'Tip' section with a lightbulb icon.

💡 Tip

If you don't see suggested queries, check your browser settings or try a different browser.

Clean up resources

When you work in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

In the Azure portal, you can find and manage resources by selecting **All resources** or **Resource groups** from the left pane.

Remember that a free search service is limited to three indexes, three indexers, and three data sources. To stay under the limit, you can delete these items individually in the Azure portal.

Next step

The demo app is useful for prototyping because it simulates the end-user experience without requiring JavaScript or front-end code. As you approach the proof-of-concept stage of your

own project, review the end-to-end code samples that more closely resemble a real-world app:

[Add search to web apps](#)

Last updated on 12/05/2025

Quickstart: Create a skillset in the Azure portal

09/16/2025

ⓘ Important

The **Import data (new)** wizard now supports keyword search, which was previously only available in the **Import data** wizard. We recommend the new wizard for an improved search experience. For more information about how we're consolidating the wizards, see [Import data wizards in the Azure portal](#).

In this quickstart, you learn how a skillset in Azure AI Search adds optical character recognition (OCR), image analysis, language detection, text merging, and entity recognition to generate text-searchable content in an index.

You can run the **Import data (new)** wizard in the Azure portal to apply skills that create and transform textual content during indexing. The input is your raw data, usually blobs in Azure Storage. The output is a searchable index containing AI-generated image text, captions, and entities. You can then query generated content in the Azure portal using [Search explorer](#).

Before you run the wizard, you create a few resources and upload sample files.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription. You can use a free service for this quickstart.
- An [Azure Storage account](#). Use Azure Blob Storage on a standard performance (general-purpose v2) account. To avoid bandwidth charges, use the same region as Azure AI Search.

ⓘ Note

This quickstart uses [Azure AI services](#) for AI enrichment. Because the workload is small, Azure AI services is tapped behind the scenes for free processing up to 20 transactions. Therefore, you don't need to create an Azure AI services multi-service resource.

Prepare sample data

In this section, you create an Azure Storage container to store sample data consisting of various file types, including images and application files that aren't full-text searchable in their native formats.

To prepare the sample data for this quickstart:

1. Sign in to the [Azure portal](#) and select your Azure Storage account.
2. From the left pane, select **Data storage > Containers**.
3. Create a container, and then upload the [sample data](#) to the container.

Run the wizard

To run the wizard:

1. Sign in to the [Azure portal](#) and select your search service.
2. On the **Overview** page, select **Import data (new)**.



3. Select **Azure Blob Storage** for the data source.

A screenshot of the 'Choose a data source' dialog box. It lists various data sources with their icons and descriptions. The 'Azure Blob Storage' option is highlighted with a red box. Other options include 'Azure Data Lake Storage Gen2', 'Azure Cosmos DB', 'Azure SQL Database', 'Fabric OneLake files (Preview)', 'SharePoint', 'OneDrive', 'OneDrive for Business', 'Azure File Storage', 'Azure Queues', 'Service Bus', 'Amazon S3', 'Dropbox', and 'SFTP - SSH'. A search bar at the top says 'Filter by name...'. A magnifying glass icon is in the bottom right corner of the dialog.

4. Select **Keyword search**.

What scenario are you targeting?



Keyword search

Ingest text for keyword search. Optionally, add AI skills to extract, infer, or create new searchable content.



RAG

Ingest text and simple images containing text (via OCR) to enable AI-powered answers.



Multimodal RAG

Ingest text and complex images (diagrams, charts, workflows) where interpreting visual elements is necessary for insights.



Step 1: Create a data source

Azure AI Search requires a connection to a data source for content ingestion and indexing. In this case, the data source is your Azure Storage account.

To create the data source:

1. On the [Connect to your data](#) page, select your Azure subscription.
2. Select your storage account, and then select the container you created.

Configure your Azure Blob Storage

Connect to Azure Blob Storage to access your structured and unstructured data files, including PDFs. [Learn more](#)

Subscription *

my-subscription



Storage account *

mystorageaccount



Blob container * ⓘ

my-container



Blob folder ⓘ

your/folder/here

Parsing mode

Default



Enable deletion tracking. ⓘ



Authenticate using managed identity. [Learn more](#)

3. Select **Next**.

If you get `Error detecting index schema from data source`, the indexer that powers the wizard can't connect to your data source. The data source most likely has security protections. Try the following solutions, and then rerun the wizard.

Expand table

Security feature	Solution
Resource requires Azure roles, or its access keys are disabled.	Connect as a trusted service or connect using a managed identity .

Security feature	Solution
Resource is behind an IP firewall.	Create an inbound rule for Azure AI Search and the Azure portal.
Resource requires a private endpoint connection.	Connect over a private endpoint.

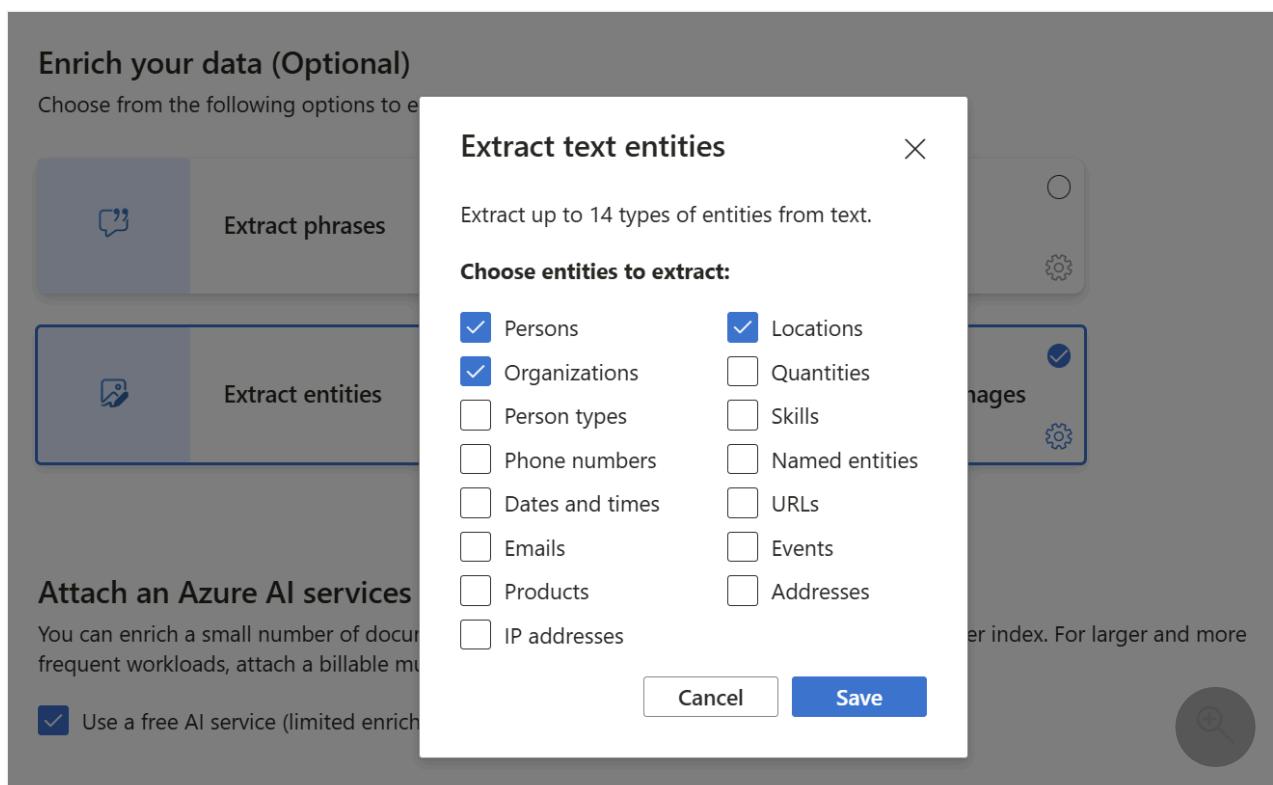
Step 2: Add cognitive skills

The next step is to configure AI enrichment to invoke OCR, image analysis, and entity recognition.

OCR and image analysis are available for blobs in Azure Blob Storage and Azure Data Lake Storage (ADLS) Gen2 and for image content in Microsoft OneLake. Images can be standalone files or embedded images in a PDF or other files.

To add the skills:

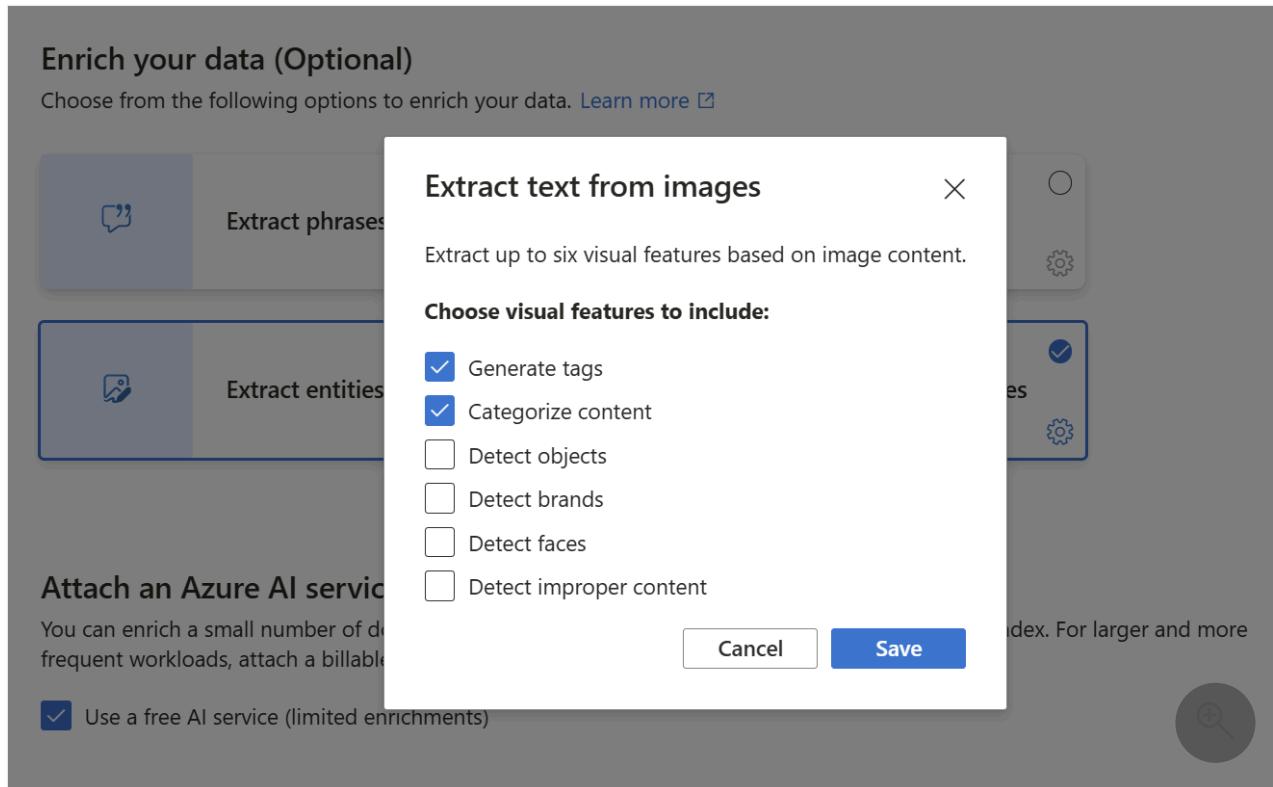
1. Select **Extract entities**, and then select the gear icon.
2. Select and save the following checkboxes:
 - **Persons**
 - **Locations**
 - **Organizations**



3. Select **Extract text from images**, and then select the gear icon.

4. Select and save the following checkboxes:

- **Generate tags**
- **Categorize content**



5. Leave the **Use a free AI service (limited enrichments)** checkbox selected.

The sample data consists of 14 files, so the free allotment of 20 transactions on Azure AI services is sufficient.

6. Select **Next**.

Step 3: Configure the index

An index contains your searchable content. The wizard can usually create the schema by sampling the data source. In this step, you review the generated schema and potentially revise any settings.

For this quickstart, the wizard sets reasonable defaults:

- Default fields are based on metadata properties of existing blobs and new fields for the enrichment output, such as `persons`, `locations`, and `organizations`. Data types are inferred from metadata and by data sampling.

Preview index fields

Review the fields that will be included in your index. Index fields can obtain values from your data source and from skill outputs if you added enrichments in the previous step. [Learn more](#)

Source column	Target index field name	Target index field type
<content + normalized_images>	→ merged_content	Edm.String
metadata_storage_name	→ title	Edm.String
metadata_storage_path	→ id	Edm.String
merged_content	→ persons	Collection(Edm.String)
merged_content	→ locations	Collection(Edm.String)
merged_content	→ organizations	Collection(Edm.String)
normalized_images	→ text	Collection(Edm.String)
normalized_images	→ layoutText	Collection(Edm.String)
normalized_images	→ tags	Collection(Edm.ComplexType)
normalized_images	→ name	Edm.String
normalized_images	→ hint	Edm.String
normalized_images	→ confidence	Edm.Double

- Default document key is `metadata_storage_path`, which is selected because the field contains unique values.
- Default field attributes are based on the skills you selected. For example, fields created by the Entity Recognition skill (`persons`, `locations`, and `organizations`) are **Retrievable**, **Filterable**, **Facetable**, and **Searchable**. To view and change these attributes, select a field, and then select **Configure field**.

Retrievable fields can be returned in results, while **Searchable** fields support full-text search. Use **Filterable** if you want to use fields in a filter expression.

Marking a field as **Retrievable** doesn't mean that the field *must* appear in search results. You can control which fields are returned by using the `select` query parameter.

After you review the index schema, select **Next**.

Step 4: Skip advanced settings

The wizard offers advanced settings for semantic ranking and index scheduling, which are beyond the scope of this quickstart. Skip this step by selecting **Next**.

Step 5: Review and create objects

The last step is to review your configuration and create the index, indexer, and data source on your search service. The indexer automates the process of extracting content from your data source, loading the index, and driving skillset execution.

To review and create the objects:

1. Accept the default Objects name prefix.

2. Review the object configurations.

Based on your configuration, the wizard will create an index, an indexer, a data source, and a skillset on your search service. You can view and manage these objects after they're created, but their names and many other properties are fixed. To customize the name, change the object name prefix.

Objects name prefix

search-1757520343372

Review your configuration

Data source

Azure Blob Storage - my-container

AI enrichments

Disabled

Semantic ranker

Enabled

Indexer run schedule

Once



AI enrichment, semantic ranker, and indexer scheduling are either disabled or set to their default values because you skipped their wizard steps.

3. Select **Create** to simultaneously create the objects and run the indexer.

Monitor status

You can monitor the creation of the indexer in the Azure portal. Skills-based indexing takes longer than text-based indexing, especially OCR and image analysis.

To monitor the progress of the indexer:

1. From the left pane, select **Indexers**.
2. Select your indexer from the list.
3. Select **Success** (or **Failed**) to view execution details.

my-indexer ...

Indexer

Run Reset Save Refresh Edit JSON Delete

Execution history Settings

Number of recent runs to show: 10

16
12
8
4
0

9/9/2025, 2:59:15 PM

Status Last run Duration Docs succeeded Errors or warnings

Success	9/9/2025, 2:59:15...	13 s	14	0/2
---------	----------------------	------	----	-----

Success Failed In Progress Reset Partial Success

Search icon

In this quickstart, there are a few warnings, including `Could not execute skill because one or more skill input was invalid.` This warning tells you that a PNG file in the data source doesn't provide a text input to Entity Recognition. It occurs because the upstream OCR skill didn't recognize any text in the image and couldn't provide a text input to the downstream Entity Recognition skill.

Warnings are common in skillset execution. As you become familiar with how skills iterate over your data, you might begin to notice patterns and learn which warnings are safe to ignore.

Query in Search explorer

To query your index:

1. From the left pane, select **Indexes**.
2. Select your index from the list. If the index has zero documents or storage, wait for the Azure portal to refresh.
3. On the **Search explorer** tab, enter a search string, such as `satya nadella`.

The search bar accepts keywords, quote-enclosed phrases, and operators. For example: `"Satya Nadella" +"Bill Gates" +"Steve Ballmer"`

Results are returned as verbose JSON, which can be hard to read, especially in large documents. Here are tips for searching in this tool:

- Switch to the JSON view to specify parameters that shape results.
- Add `select` to limit the fields in results.
- Add `count` to show the number of matches.
- Use Ctrl-F to search within the JSON for specific properties or terms.

my-index

The screenshot shows the Azure Search interface for the index "my-index". At the top, there are navigation links: Save, Discard, Refresh, Create demo app, Edit JSON, Delete, and Encryption. Below that, it displays the number of documents (14), total storage (395 KB), vector index quota usage (0 Bytes), and max storage (160 GB). The "Search explorer" tab is selected. On the right, there's a "View" dropdown menu with options: Query view, Image view, and JSON view, with "JSON view" highlighted by a red box. A search bar at the top right contains the query "satya nadella". The results section shows a JSON response with 3 items found. The JSON code in the editor is:

```
1 {  
2   "search": "\"Satya Nadella\" +\"Bill Gates\" +\"Steve Ballmer\"",  
3   "count": true,  
4   "select": "merged_content, persons"  
5 }
```

The search results table has columns for @odata.context, @odata.count, and value. The value column shows the merged content and a list of persons: Bill Gates, Steve Ballmer, Satya Nadella, and Reid Hoffman.

Here's some JSON you can paste into the view:

A JSON query editor window is shown. The JSON code is identical to the one in the previous screenshot:

```
{  
  "search": "\"Satya Nadella\" +\"Bill Gates\" +\"Steve Ballmer\"",  
  "count": true,  
  "select": "merged_content, persons"  
}
```

💡 Tip

Query strings are case sensitive, so if you get an "unknown field" message, check **Fields** or **Index Definition (JSON)** to verify the name and case.

Takeaways

You've created your first skillset and learned the basic steps of skills-based indexing.

Some key concepts that we hope you picked up include the dependencies. A skillset is bound to an indexer, and indexers are Azure and source-specific. Although this quickstart uses Azure

Blob Storage, other Azure data sources are available. For more information, see [Indexers in Azure AI Search](#).

Another important concept is that skills operate over content types, and when you use heterogeneous content, some inputs are skipped. Also, large files or fields might exceed the indexer limits of your service tier. It's normal to see warnings when these events occur.

The output is routed to a search index, and there's a mapping between name-value pairs created during indexing and individual fields in your index. Internally, the wizard sets up [an enrichment tree](#) and defines a [skillset](#), establishing the order of operations and general flow. These steps are hidden in the wizard, but when you start writing code, these concepts become important.

Finally, you learned that you can verify content by querying the index. Ultimately, Azure AI Search provides a searchable index that you can query using either [simple](#) or [fully extended query syntax](#). An index containing enriched fields is like any other. You can incorporate standard or [custom analyzers](#), [scoring profiles](#), [synonyms](#), [faceted navigation](#), geo-search, and other Azure AI Search features.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal by selecting [All resources](#) or [Resource groups](#) from the left pane.

If you used a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the Azure portal to stay under the limit.

Next step

You can use the Azure portal, REST APIs, or an Azure SDK to create skillsets. Try the REST APIs by using a REST client and more sample data:

[Tutorial: Use skillsets to generate searchable content in Azure AI Search](#)

Quickstart: Create a knowledge store in the Azure portal

(!) Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in [agentic retrieval](#) workflows.

In this quickstart, you create a [knowledge store](#) that serves as a repository for output generated from an [AI enrichment pipeline](#) in Azure AI Search. A knowledge store makes generated content available in Azure Storage for workloads other than search.

First, you set up sample data in Azure Storage. Next, you run the **Import data** wizard to create an enrichment pipeline that also generates a knowledge store. The knowledge store contains original source content pulled from the data source (customer reviews of a hotel), plus AI-generated content that includes a sentiment label, key phrase extraction, and text translation of non-English customer comments.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription. For this quickstart, you can use a free service.
- An Azure Storage account. [Create an account](#) or [find an existing account](#). The account type must be **StorageV2 (general purpose V2)**.
- Sample data hosted in Azure Storage:
 - [Download HotelReviews_Free.csv](#), which contains 19 pieces of customer feedback about a single hotel (originates from Kaggle.com). This CSV is in a repo with other sample data. If you don't want the whole repo, copy the raw content and paste it into a spreadsheet app on your device.
 - [Upload the file to a blob container](#) in Azure Storage.

(!) Note

This quickstart uses [Foundry Tools](#) for AI enrichment. Because the workload is so small, Foundry Tools is tapped behind the scenes for free processing for up to 20 transactions. This means that you can complete this exercise without having to create an extra Microsoft Foundry resource.

Start the wizard

1. Sign in to the [Azure portal](#) with your Azure account.
2. [Find your search service](#) and on the Overview page, select **Import data** on the command bar to create a knowledge store in four steps.



Step 1: Create a data source

Because the data is multiple rows in one CSV file, set the *parsing mode* to get one search document for each row.

1. In **Connect to your data**, choose **Azure Blob Storage**.
2. For the **Name**, enter "hotel-reviews-ds".
3. For **Data to extract**, choose **Content and Metadata**.
4. For **Parsing mode**, select **Delimited text**, and then select the **First Line Contains Header** checkbox. Make sure the **Delimiter character** is a comma (,).
5. In **Connection String**, choose an existing connection if the storage account is in the same subscription. Otherwise, paste in a connection string to your Azure Storage account.

A connection string can be full access, having the following format:

```
DefaultEndpointsProtocol=https;AccountName=<YOUR-ACCOUNT-NAME>;AccountKey=<YOUR-  
ACCOUNT-KEY>;EndpointSuffix=core.windows.net
```

Or, a connection string can reference a managed identity, assuming it's [configured and assigned a role](#) in Azure Storage: `ResourceId=/subscriptions/<YOUR-SUBSCRIPTION-ID>/resourceGroups/<YOUR-RESOURCE-GROUP-NAME>/providers/Microsoft.Storage/storageAccounts/<YOUR-ACCOUNT-NAME>;`

6. In **Containers**, enter the name of the blob container holding the data ("hotel-reviews").

Your page should look similar to the following screenshot.



7. Continue to the next page.

Step 2: Add skills

In this wizard step, add skills for AI enrichment. The source data consists of customer reviews in English and French. Skills that are relevant for this data set include key phrase extraction, sentiment detection, and text translation. In a later step, these enrichments are "projected" into a knowledge store as Azure tables.

1. Expand **Attach Foundry Tools**. **Free (Limited enrichments)** is selected by default. You can use this resource because the number of records in HotelReviews-Free.csv is 19 and this free resource allows up to 20 transactions a day.
2. Expand **Add enrichments**.
3. For **Skillset name**, enter "hotel-reviews-ss".
4. For **Source data field**, select **reviews_text**.
5. For **Enrichment granularity level**, select **Pages (5000 characters chunks)**.
6. For **Text Cognitive Skills**, select the following skills:
 - Extract key phrases
 - Translate text
 - Language detection
 - Detect sentiment

Your page should look like the following screenshot:

Skillset name * ⓘ

Enable OCR and merge all text into **merged_content** field ⓘ

Source data field *

<input type="text" value="reviews_text"/>	▼
---	---

Enrichment granularity level ⓘ

<input type="text" value="Pages (5000 characters chunks)"/>	▼
---	---

Checked items below require a field name.

Parameter	Field name
<input type="checkbox"/> Text Cognitive Skills	
<input type="checkbox"/> Extract people names	people
<input type="checkbox"/> Extract organization names	organizations
<input type="checkbox"/> Extract location names	locations
<input checked="" type="checkbox"/> Extract key phrases	<input type="text" value="keyphrases"/>
<input checked="" type="checkbox"/> Detect language	language
<input checked="" type="checkbox"/> Translate text	Target Language <input type="button" value="English"/> translated_text
<input type="checkbox"/> Extract personally identifiable infor...	pii_entities
<input checked="" type="checkbox"/> Detect sentiment	sentiment

7. Scroll down and expand **Save enrichments to knowledge store**.

8. Select **Choose an existing connection** and then select an Azure Storage account. The **Containers** page appears so that you can create a container for projections. We recommend adopting a prefix naming convention, such as "kstore-hotel-reviews" to distinguish between source content and knowledge store content.

9. Returning to the Import data wizard, select the following **Azure table projections**. The wizard always offers the **Documents** projection. Other projections are offered depending on the skills you select (such as **Key phrases**), or the enrichment granularity (**Pages**):

- **Documents**
- **Pages**
- **Key phrases**

The following screenshot shows the table projection selections in the wizard.



10. Continue to the next page.

Step 3: Configure the index

In this wizard step, configure an index for optional full-text search queries. You don't need a search index for a knowledge store, but the indexer requires one in order to run.

In this step, the wizard samples your data source to infer fields and data types. You only need to select the attributes for your desired behavior. For example, the **Retrievable** attribute allows the search service to return a field value, while the **Searchable** attribute enables full text search on the field.

1. For **Index name**, enter "hotel-reviews-idx".
2. For attributes, accept the default selections: **Retrievable** and **Searchable** for the new fields that the pipeline is creating.

Your index should look similar to the following image. Because the list is long, not all fields are visible in the image.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACETABLE	SEARCHABLE	ANALYZER
keyphrases	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...
translated_text	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Lucene
sentiment	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

3. Continue to the next page.

Step 4: Configure and run the indexer

In this wizard step, configure an indexer that pulls together the data source, skillset, and the index you defined in the previous wizard steps.

1. For **Name**, enter "hotel-reviews-idxr".
2. For **Schedule**, keep the default **Once**.
3. Select **Submit** to run the indexer. Data extraction, indexing, application of cognitive skills all happen in this step.

Step 5: Check status

In the **Overview** page, open the **Indexers** tab in the middle of the page, and then select **hotels-reviews-idxr**. Within a minute or two, status should progress from "In progress" to "Success" with zero errors and warnings.

Check tables in Azure portal

1. In the Azure portal, [open the Storage account](#) used to create the knowledge store.

2. In the storage account's left pane, select **Storage browser** to view the new tables.

You should see three tables, one for each projection that was offered in the "Save enrichments" section of the "Add enrichments" page.

- "hotelReviewssDocuments" contains all of the first-level nodes of a document's enrichment tree that aren't collections.
- "hotelReviewssKeyPhrases" contains a long list of just the key phrases extracted from all reviews. Skills that output collections (arrays), such as key phrases and entities, send output to a standalone table.
- "hotelReviewssPages" contains enriched fields created over each page that was split from the document. In this skillset and data source, page-level enrichments consisting of sentiment labels and translated text. A pages table (or a sentences table if you specify that particular level of granularity) is created when you choose "pages" granularity in the skillset definition.

All of these tables contain ID columns to support table relationships in other tools and apps. When you open a table, scroll past these fields to view the content fields added by the pipeline.

In this quickstart, the table for "hotelReviewssPages" should look similar to the following screenshot:

The screenshot shows the Azure Storage browser (preview) interface. The left sidebar shows storage accounts (blobstorage2), favorites, recently viewed items (blobstorage2), blob containers, file shares, queues, and tables. The 'Tables' section is expanded, and 'hotelReviewsPages' is selected. The main area shows the 'hotelReviewsPages' table with 19 items. The table has columns: Pagesid, languageCode, sentimentScore, and translatedText. The rows show data corresponding to the table description in the text above.

Pagesid	languageCode	sentimentScore	translatedText
min walk al...	en	mixed	Pleasant 10 min walk al...
with very fri...	en	positive	Nice hotel , with very fri...
nin+ walk to...	en	positive	It was a 10 min+ walk to...

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the Azure portal to stay under the limit.

Tip

If you want to repeat this exercise or try a different AI enrichment walkthrough, delete the **hotel-reviews-idxr** indexer and the related objects to recreate them. Deleting the indexer resets the free daily transaction counter to zero.

Next step

Now that you've been introduced to a knowledge store, take a closer look at each step by completing the REST API walkthrough. The walkthrough explains tasks that the wizard handled internally.

[Create a knowledge store using REST](#)

Last updated on 11/18/2025

Quickstart: Use Search explorer to run queries in the Azure portal

In this quickstart, you learn how to use **Search explorer**, a built-in query tool in the Azure portal for running queries against an Azure AI Search index. Use this tool to test a query or filter expression or to confirm whether content exists in the index.

This quickstart uses an existing index to demonstrate Search explorer.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure AI Search service. [Create a service](#) or [find an existing service](#) in your current subscription. For this quickstart, you can use a free service.
- This quickstart uses the hotels-sample index. Follow the instructions in [this quickstart](#) to create the index.

Start Search explorer

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview**.
3. On the command bar, select **Search explorer**.



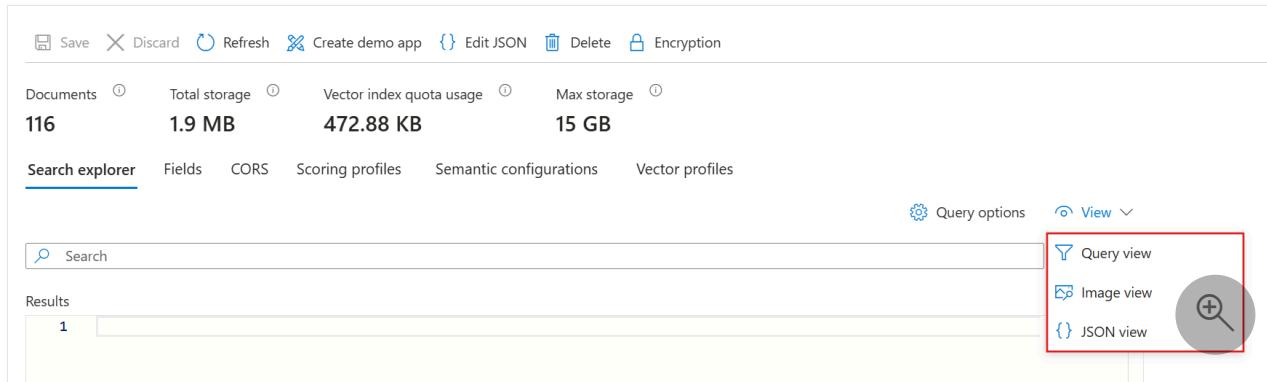
Alternatively, select the **Search explorer** tab on the index page.

Query three ways

There are three approaches to querying in Search explorer:

- Query view provides a default search bar. It accepts an empty query or free-text query with Booleans, such as `ocean view + parking`.
- Image view provides a window to browse or drag and drop PNG, JPG, or JPEG files. Unless your index has an [image vectorizer](#) and an equivalent skill, this view is unavailable.

- JSON view supports parameterized queries. Filters, orderby, select, count, searchFields, and all other parameters must be set in JSON view.



Example: Image query

Search explorer accepts images as query inputs through **Image view**, which requires that you use a supported vectorizer–skill pair. For more information, see [Configure a vectorizer in a search index](#).

The hotels-sample index isn't configured for image vectorization. If you want to run image queries, create an index as described in [Quickstart: Vector search in the Azure portal](#). The quickstart relies on text-based sample data, so you must use documents that contain images.

To run an image query, select or drag an image to the search area, and then select **Search**. Search explorer vectorizes the image and sends the vector to the search engine for query execution. The search engine returns documents that are sufficiently similar to the input image, up to the specified **k** number of results.


✓ city-skyline.p*l* X

Drag and drop an image or [Browse image file](#)

Search

Results

```
1 {  
2   "@odata.context": "https://my-search-service.search.windows.net/indexes('vector-image-search')/$metadata#docs()  
3   "@odata.count": 50,  
4   "value": [  
5     {  
6       "@search.score": 0.6021544,  
7       "chunk_id": "7731306efa5f_aHR0cHM6Ly9oYWlsZX1zdG9yYWdlLmJsb2IuY29yZS53aW5kb3dzLm51dC9tdWx0aW1vZGFsLXN1YXJjaC9BY2Nz  
8       "text_parent_id": "aHR0cHM6Ly9oYWlsZX1zdG9yYWdlLmJsb2IuY29yZS53aW5kb3dzLm51dC9tdWx0aW1vZGFsLXN1YXJjaC9BY2Nz  
9       "chunk": "transmission lines in a \nrenewable energy-rich region of New York state.12 \n\nAccelerating Sus  
10      "title": "Accelerating-Sustainability-with-AI-2025.pdf",  
11      "image_parent_id": null,  
12      "text_vector": [  
13        -0.010551453,  
14        0.012992859,  
15        0.021560669.
```



Examples: JSON queries

The following are examples of JSON queries you can run using Search explorer. To follow these examples, switch to **JSON view**. You can paste each JSON example into the text area.

Tip

The JSON view supports intellisense for parameter name completion. Place your cursor inside the JSON view and enter a space character to see a list of all query parameters. You can also enter a letter, like `s`, to see only the query parameters that begin with that letter.

Intellisense doesn't exclude invalid parameters, so use your best judgment.

Run an unspecified query

In Search explorer, POST requests are formulated internally using [Documents - Search Post \(REST API\)](#), with responses returned as verbose JSON documents.

For a first look at content, execute an empty search by selecting **Search** with no terms provided. An empty search is useful as a first query because it returns entire documents so that you can review document composition. On an empty search, there's no search score, and

documents are returned in arbitrary order (`"@search.score": 1` for all documents). By default, 50 documents are returned per search request.

Add `"count": true` to get the number of matches found in an index. On an empty search, the count is the total number of documents in the index. On a qualified search, it's the number of documents matching the query input. Recall that the service returns the top-50 matches by default, so the count might indicate more matches in the index than what's returned in the results.

Equivalent syntax for an empty search is `*` or `"search": "*"`.

JSON

```
{  
  "search": "*",
  "count": true
}
```

Results

JSON query editor

```
1  {
2    "search": "*",
3    "count": true
4  }
```

Search

Results

```
1  {
2    "@odata.context": null,
3    "@odata.count": 50,
4    "value": [
5      {
6        "@search.score": 1,
7        "HotelId": "20",
8        "HotelName": "Grand Gaming Resort",
9        "Description": "The Best Gaming Resort in the area. With elegant rooms & suites, pool, cabanas, spa, bre",
10       "Description_fr": "La meilleure station de jeux dans la région. Avec des chambres et suites élégantes, p",
11       "Category": "Resort and Spa",
12       "Tags": [
13         "continental breakfast",
14         "bar",
15         "pool"
16       ],
17       "ParkingIncluded": true,
18       "LastRenovationDate": "2021-10-31T00:00:00Z",
19       "Rating": 4.2,
20       "Location": {
21         "type": "Point",
22         "coordinates": [
23           -106.605949,
24           35.1087
25         ]
26       }
27     ]
28   }
29 }
```



Run a free-text query

Free-form search, with or without operators, is useful for simulating user-defined queries sent from a custom app to Azure AI Search. Only fields attributed as searchable in the index are

scanned for matches.

You don't need the JSON view for a free-text query, but we provide it in JSON for consistency with other examples in this article.

Notice that when you provide search criteria, such as query terms or expressions, search rank comes into play. The following example illustrates a free text search. The `@search.score` is a relevance score computed for the match using the [default scoring algorithm](#).

JSON

```
{  
  "search": "activities `outdoor pool` restaurant OR continental breakfast"  
}
```

Results

You can use Ctrl-F to search within results for specific terms of interest.

The screenshot shows a JSON query editor interface. At the top, there is a code editor window containing the following JSON:

```
1 {  
2   "search": "activities `outdoor pool` restaurant OR continental breakfast"  
3 }
```

Below the code editor is a results pane titled "Results". It displays a single JSON object with line numbers from 1 to 19. The line number 5 is highlighted with a red box, showing the field `"@search.score": 7.0059757,`. The results pane also contains a search bar with the text "outdoor pool", a status bar indicating "Aa ab .* 1 of 3", and a vertical scrollbar on the right.

Limit fields in search results

Add "[select](#)" to limit results to the explicitly named fields for more readable output in [Search explorer](#). Only fields attributed as retrievable in the index can show up in results.

JSON

```
{
  "search": "activities `outdoor pool` restaurant OR continental breakfast",
  "count": true,
  "select": "HotelId, HotelName, Tags, Description"
}
```

Results

JSON query editor

```

1  {
2    "search": "activities `outdoor pool` restaurant OR continental breakfast",
3    "count": true,
4    "select": "HotelId, HotelName, Tags, Description"
5  }
```

Search

Results

```

1  {
2    "@odata.context": "https://contoso.com/api/v1/hotels/$metadata#Collection(Hotel)",
3    "@odata.count": 34,
4    "value": [
5      {
6        "@search.score": 7.0059757,
7        "HotelId": "21",
8        "HotelName": "Good Business Hotel",
9        "Description": "1 Mile from the airport. Free WiFi, Outdoor Pool, Complimentary Airport Shuttle, 6 miles from downtown. Great for business travel and leisure. Book now!",
10       "Tags": [
11         "pool",
12         "continental breakfast",
13         "free parking"
14       ],
15     },
16     {
17       "@search.score": 6.787462,
18       "HotelId": "12",
19       "HotelName": "Winter Panorama Resort",
20       "Description": "Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, Pilates, and fitness classes available. Book now!",
21       "Tags": [
22     ]
23     }
24   ]
25 }
```

Return next batch of results

Azure AI Search returns the top-50 matches based on the search rank. The hotels-sample index only has 50 hotels, so we use a smaller number to illustrate paging. To get the next set of matching documents, append `"top": 20` and `"skip": 10` to increase the result set to 20 documents (default is 50, maximum is 1000), skipping the first 10 documents. You can check the document key (`HotelId`) to identify a document.

Recall that you need to provide search criteria, such as a query term or expression, to get ranked results. Search scores decrease the deeper you reach into search results.

JSON

```
{
  "search": "activities `outdoor pool` restaurant OR continental breakfast",
  "count": true,
  "select": "HotelId, HotelName, Tags, Description",
  "top": 20,
```

```
    "skip": 10
}
```

Results

JSON query editor

```
1 {  
2   "search": "activities `outdoor pool` restaurant OR continental breakfast",  
3   "count": true,  
4   "select": "HotelId, HotelName, Tags, Description",  
5   "top": 20,  
6   "skip": 10  
7 }
```

Search

Results

```
1 {  
2   "@odata.context":  
3   "@odata.count": 34,  
4   "value": [  
5     {  
6       "@search.score": 3.872953,  
7       "HotelId": "50",  
8       "HotelName": "Head Wind Resort",  
9       "Description": "The best of old town hospitality combined with views of the river and cool breezes off the sea.",  
10      "Tags": [  
11        "coffee in lobby",  
12        "free wifi",  
13        "view"  
14      ]  
15    },  
16    {  
17      "@search.score": 3.8503008,  
18      "HotelId": "16",  
19      "HotelName": "Double Sanctuary Resort",  
20      "Description": "5 star Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Traveler's Choice.",  
21      "Tags": [  
22        "view",  
23        "spa",  
24        "pool",  
25        "dinner",  
26        "spa",  
27        "pool",  
28        "dinner",  
29        "spa",  
30        "pool",  
31        "dinner",  
32        "spa",  
33        "pool",  
34        "dinner",  
35        "spa",  
36        "pool",  
37        "dinner",  
38        "spa",  
39        "pool",  
40        "dinner",  
41        "spa",  
42        "pool",  
43        "dinner",  
44        "spa",  
45        "pool",  
46        "dinner",  
47        "spa",  
48        "pool",  
49        "dinner",  
50        "spa",  
51        "pool",  
52        "dinner",  
53        "spa",  
54        "pool",  
55        "dinner",  
56        "spa",  
57        "pool",  
58        "dinner",  
59        "spa",  
60        "pool",  
61        "dinner",  
62        "spa",  
63        "pool",  
64        "dinner",  
65        "spa",  
66        "pool",  
67        "dinner",  
68        "spa",  
69        "pool",  
70        "dinner",  
71        "spa",  
72        "pool",  
73        "dinner",  
74        "spa",  
75        "pool",  
76        "dinner",  
77        "spa",  
78        "pool",  
79        "dinner",  
80        "spa",  
81        "pool",  
82        "dinner",  
83        "spa",  
84        "pool",  
85        "dinner",  
86        "spa",  
87        "pool",  
88        "dinner",  
89        "spa",  
90        "pool",  
91        "dinner",  
92        "spa",  
93        "pool",  
94        "dinner",  
95        "spa",  
96        "pool",  
97        "dinner",  
98        "spa",  
99        "pool",  
100       "dinner",  
101      ]  
102    }  
103  ]
```



Filter expressions (greater than, less than, equal to)

Use the `filter` parameter to specify inclusion or exclusion criteria. The field must be attributed as filterable in the index. This example searches for ratings greater than four:

JSON

```
{  
  "search": "activities `outdoor pool` restaurant OR continental breakfast",  
  "count": true,  
  "select": "HotelId, HotelName, Tags, Description, Rating",  
  "filter": "Rating gt 4"  
}
```

Results

JSON query editor

```

1  {
2    "search": "activities `outdoor pool` restaurant OR continental breakfast",
3    "count": true,
4    "select": "HotelId, HotelName, Tags, Description, Rating",
5    "filter": "Rating gt 4"
6  }

```

Search

Results

```

6  "@search.score": 6.787462,
7  "HotelId": "12",
8  "HotelName": "Winter Panorama Resort",
9  "Description": "Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, i
10 "Tags": [
11   "restaurant",
12   "bar",
13   "pool"
14 ],
15 "Rating": 4.5
16 },
17 {
18   "@search.score": 5.4877596,
19   "HotelId": "18",
20   "HotelName": "Ocean Water Resort & Spa",
21   "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near
22   "Tags": [
23     "view",
24     "pool",
25     "restaurant"
26 ],

```



Sort results

Add [orderby](#) to sort results by another field besides search score. The field must be attributed as sortable in the index. In situations where the filtered value is identical (for example, same price), the order is arbitrary, but you can add more criteria for deeper sorting. Here's an example expression you can use to test this out:

JSON

```
{
  "search": "activities `outdoor pool` restaurant OR continental breakfast",
  "count": true,
  "select": "HotelId, HotelName, Tags, Description, Rating, LastRenovationDate",
  "filter": "Rating gt 4",
  "orderby": "LastRenovationDate desc"
}
```

Results

JSON query editor

```

1   {
2     "search": "activities `outdoor pool` restaurant OR continental breakfast",
3     "count": true,
4     "select": "HotelId, HotelName, Tags, Description, Rating, LastRenovationDate",
5     "filter": "Rating gt 4",
6     "orderby": "LastRenovationDate desc"
7   }

```

Search

Results

```

6   "@search.score": 0.9842338,
7   "HotelId": "14",
8   "HotelName": "Twin Vortex Hotel",
9   "Description": "New experience in the making. Be the first to experience the luxury of the Twin Vortex. Re",
10  "Tags": [
11    "bar",
12    "restaurant",
13    "concierge"
14  ],
15  "LastRenovationDate": "2023-11-14T00:00:00Z",
16  "Rating": 4.4
17 },
18 {
19   "@search.score": 6.787462,
20   "HotelId": "12",
21   "HotelName": "Winter Panorama Resort",
22   "Description": "Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, a",
23   "Tags": [
24     "restaurant",
25     "bar",
26     "pool"

```

Takeaways

In this quickstart, you used **Search explorer** to query an index using the REST API.

- Results are returned as verbose JSON documents so that you can view the construction and content of each document in its entirety. The `select` parameter in a query expression limits which fields are returned.
- Search results are composed of all fields attributed as retrievable in the index. Select the **Fields** tab to review attributes.
- Keyword search, similar to what you might enter in a commercial web browser, is useful for testing an end-user experience. For example, assuming the hotels-sample index, you can enter `"activities 'outdoor pool' restaurant OR continental breakfast"`, and then you can use Ctrl-F to find terms within the search results.
- Query and filter expressions are articulated in a syntax implemented by Azure AI Search. The default is a [simple syntax](#), but you can optionally use [full Lucene](#) for more powerful queries. [Filter expressions](#) are articulated in an OData syntax.

Clean up resources

When you work in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money.

You can delete resources individually or delete the resource group to delete the entire set of resources.

In the Azure portal, you can find and manage resources by selecting **All resources** or **Resource groups** from the left pane.

Remember that a free search service is limited to three indexes, three indexers, and three data sources. To stay under the limit, you can delete these items individually in the Azure portal.

Next step

To learn more about query structures and syntax, use a REST client to create query expressions that use more parts of the REST API. [Documents - Search Post \(REST API\)](#) is especially helpful for learning and exploration.

[Quickstart: Full-text search](#)

Last updated on 12/04/2025

Quickstart: Deploy Azure AI Search using an Azure Resource Manager template

Article • 03/04/2025

In this quickstart, you use an Azure Resource Manager (ARM) template to deploy an Azure AI Search service in the Azure portal.

An [Azure Resource Manager template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. You describe your intended deployment without writing the sequence of programming commands to create the deployment.

Only those properties included in the template are used in the deployment. If more customization is required, such as [setting up network security](#), you can update the service as a post-deployment task. To customize an existing service with the fewest steps, use [Azure CLI](#) or [Azure PowerShell](#). If you're evaluating preview features, use the [Management REST API](#).

Assuming your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal.



Deploy to Azure



Prerequisites

If you don't have an Azure subscription, create a [free account](#) before you begin.

Review the template

The template used in this quickstart is from [Azure Quickstart Templates](#).

JSON

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2019-04-  
  01/deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "metadata": {
```

```
"_generator": {
    "name": "bicep",
    "version": "0.5.6.12127",
    "templateHash": "11257266040777038564"
}
},
"parameters": {
    "name": {
        "type": "string",
        "maxLength": 60,
        "minLength": 2,
        "metadata": {
            "description": "Service name must only contain lowercase letters, digits or dashes, cannot use dash as the first two or last one characters, cannot contain consecutive dashes, and is limited between 2 and 60 characters in length."
        }
    },
    "sku": {
        "type": "string",
        "defaultValue": "standard",
        "metadata": {
            "description": "The pricing tier of the search service you want to create (for example, basic or standard)."
        }
    },
    "allowedValues": [
        "free",
        "basic",
        "standard",
        "standard2",
        "standard3",
        "storage_optimized_11",
        "storage_optimized_12"
    ]
},
"replicaCount": {
    "type": "int",
    "defaultValue": 1,
    "maxValue": 12,
    "minValue": 1,
    "metadata": {
        "description": "Replicas distribute search workloads across the service. You need at least two replicas to support high availability of query workloads (not applicable to the free tier)."
    }
},
"partitionCount": {
    "type": "int",
    "defaultValue": 1,
    "allowedValues": [
        1,
        2,
        3,
        4,
        6,
    ]
}
```

```

    12
  ],
  "metadata": {
    "description": "Partitions allow for scaling of document count as well as faster indexing by sharding your index over multiple search units."
  }
},
"hostingMode": {
  "type": "string",
  "defaultValue": "default",
  "allowedValues": [
    "default",
    "highDensity"
  ],
  "metadata": {
    "description": "Applicable only for SKUs set to standard3. You can set this property to enable a single, high density partition that allows up to 1000 indexes, which is much higher than the maximum indexes allowed for any other SKU."
  }
},
"location": {
  "type": "string",
  "defaultValue": "[resourceGroup().location]",
  "metadata": {
    "description": "Location for all resources."
  }
}
},
"resources": [
{
  "type": "Microsoft.Search/searchServices",
  "apiVersion": "2020-08-01",
  "name": "[parameters('name')]",
  "location": "[parameters('location')]",
  "sku": {
    "name": "[parameters('sku')]"
  },
  "properties": {
    "replicaCount": "[parameters('replicaCount')]",
    "partitionCount": "[parameters('partitionCount')]",
    "hostingMode": "[parameters('hostingMode')]"
  }
}
]
}

```

The Azure resource defined in this template:

- [Microsoft.Search/searchServices](#): create an Azure AI Search service

Deploy the template

Select the following image to sign in to Azure and open a template. The template creates an Azure AI Search resource.



The Azure portal displays a form that allows you to easily provide parameter values. Some parameters are prefilled with the default values from the template. Provide your subscription, resource group, location, and service name. If you want to use Azure AI services in an [AI enrichment](#) pipeline, for example to analyze binary image files for text, choose a location that offers both Azure AI Search and Azure AI services. Both services are required to be in the same region for AI enrichment workloads. After you complete the form, agree to the terms and conditions and then select the purchase button to complete your deployment.

Azure Search service

Azure quickstart template

TEMPLATE



101-azure-search-create

1 resource



Edit template



Edit param...



Learn more

BASICS

Subscription *



Resource group *



Create new

Location *



SETTINGS

Name * ⓘ

Sku ⓘ



Replica Count ⓘ

Partition Count ⓘ



Hosting Mode ⓘ



Location ⓘ

TERMS AND CONDITIONS

[Template information](#) | [Azure Marketplace Terms](#) | [Azure Marketplace](#)

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated with the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

 I agree to the terms and conditions stated above[Purchase](#)

Review deployed resources

When your deployment is complete, you can access your new resource group and new search service in the Azure portal.

Clean up resources

Other Azure AI Search quickstarts and tutorials build upon this quickstart. If you plan to continue on to work with subsequent quickstarts and tutorials, you may wish to leave this resource in place. When no longer needed, you can delete the resource group, which deletes the Azure AI Search service and related resources.

Related content

In this quickstart, you created an Azure AI Search service using an ARM template and then validated the deployment. To learn more about Azure AI Search and Azure Resource Manager, see the following articles:

- [What is Azure AI Search?](#)
- [Quickstart: Create a search index in the Azure portal](#)
- [Quickstart: Create a demo search app in the Azure portal](#)
- [Quickstart: Create a skillset in the Azure portal](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Deploy Azure AI Search using Bicep

Article • 03/04/2025

In this quickstart, you use a Bicep file to deploy an Azure AI Search service in the Azure portal.

Bicep is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse. Bicep offers the best authoring experience for your infrastructure-as-code solutions in Azure.

Only those properties included in the template are used in the deployment. If more customization is required, such as [setting up network security](#), you can update the service as a post-deployment task. To customize an existing service with the fewest steps, use [Azure CLI](#) or [Azure PowerShell](#). If you're evaluating preview features, use the [Management REST API](#).

💡 Tip

For an alternative Bicep template that deploys Azure AI Search with a pre-configured indexer to Cosmos DB for NoSQL, see [Bicep deployment of Azure AI Search](#). There's no bicep template support for Azure AI Search data plane operations like creating an index, but you can add a module that calls REST APIs. The sample includes a module that creates an index, data source connector, and an indexer that refreshes from Cosmos DB at 5-minute intervals.

Prerequisites

If you don't have an Azure subscription, create a [free account](#) before you begin.

Review the Bicep file

The Bicep file used in this quickstart is from [Azure Quickstart Templates](#).

Bicep

```
@description('Service name must only contain lowercase letters, digits or dashes, cannot use dash as the first two or last one characters, cannot contain consecutive dashes, and is limited between 2 and 60 characters in')
```

```
length.')
@minLength(2)
@maxLength(60)
param name string

@allowed([
    'free'
    'basic'
    'standard'
    'standard2'
    'standard3'
    'storage_optimized_l1'
    'storage_optimized_l2'
])
@description('The pricing tier of the search service you want to create (for example, basic or standard).')
param sku string = 'standard'

@description('Replicas distribute search workloads across the service. You need at least two replicas to support high availability of query workloads (not applicable to the free tier).')
@minValue(1)
@maxValue(12)
param replicaCount int = 1

@description('Partitions allow for scaling of document count as well as faster indexing by sharding your index over multiple search units.')
@allowed([
    1
    2
    3
    4
    6
    12
])
param partitionCount int = 1

@description('Applicable only for SKUs set to standard3. You can set this property to enable a single, high density partition that allows up to 1000 indexes, which is much higher than the maximum indexes allowed for any other SKU.')
@allowed([
    'default'
    'highDensity'
])
param hostingMode string = 'default'

@description('Location for all resources.')
param location string = resourceGroup().location

resource search 'Microsoft.Search/searchServices@2020-08-01' = {
    name: name
    location: location
    sku: {
        name: sku
    }
}
```

```
    }
  properties: {
    replicaCount: replicaCount
    partitionCount: partitionCount
    hostingMode: hostingMode
  }
}
```

The Azure resource defined in this Bicep file:

- [Microsoft.Search/searchServices](#): create an Azure AI Search service

Deploy the Bicep file

1. Save the Bicep file as **main.bicep** to your local computer.
2. Deploy the Bicep file using either Azure CLI or Azure PowerShell.

CLI

Azure CLI

```
az group create --name exampleRG --location eastus
az deployment group create --resource-group exampleRG --template-
file main.bicep --parameters serviceName=<service-name>
```

ⓘ Note

Replace **<service-name>** with the name of the Search service. The service name must only contain lowercase letters, digits, or dashes. You can't use a dash as the first two characters or the last character. The name has a minimum length of 2 characters and a maximum length of 60 characters.

When the deployment finishes, you should see a message indicating the deployment succeeded.

Review deployed resources

Use the Azure portal, Azure CLI, or Azure PowerShell to list the deployed resources in the resource group.

CLI

Azure CLI

```
az resource list --resource-group exampleRG
```

Clean up resources

Azure AI Search is a billable resource. If it's no longer needed, delete it from your subscription to avoid charges. You can use the Azure portal, Azure CLI, or Azure PowerShell to delete the resource group and its resources.

CLI

Azure CLI

```
az group delete --name exampleRG
```

Related content

In this quickstart, you created an Azure AI Search service using a Bicep file and then validated the deployment. To learn more about Azure AI Search and Azure Resource Manager, see the following articles:

- [What is Azure AI Search?](#)
- [Quickstart: Create a search index in the Azure portal](#)
- [Quickstart: Create a demo search app in the Azure portal](#)
- [Quickstart: Create a skillset in the Azure portal](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Deploy Azure AI Search service using Terraform

05/29/2025

This article shows how to use Terraform to create an [Azure AI Search service](#) using [Terraform](#).

[Terraform](#)  enables the definition, preview, and deployment of cloud infrastructure. Using Terraform, you create configuration files using [HCL syntax](#)  . The HCL syntax allows you to specify the cloud provider - such as Azure - and the elements that make up your cloud infrastructure. After you create your configuration files, you create an *execution plan* that allows you to preview your infrastructure changes before they're deployed. Once you verify the changes, you apply the execution plan to deploy the infrastructure.

In this article, you learn how to:

- ✓ Create a random pet name for the Azure resource group name using [random_pet](#) 
- ✓ Create an Azure resource group using [azurerm_resource_group](#) 
- ✓ Create a random string using [random_string](#) 
- ✓ Create an Azure AI Search service using [azurerm_search_service](#) 

Prerequisites

- [Install and configure Terraform](#)

Implement the Terraform code

Note

See more [articles and sample code showing how to use Terraform to manage Azure resources](#)

1. Create a directory in which to test and run the sample Terraform code and make it the current directory.
2. Create a file named `main.tf` and insert the following code:

Terraform

```
resource "random_pet" "rg_name" {
    prefix = var.resource_group_name_prefix
}
```

```

resource "azurerm_resource_group" "rg" {
  name      = random_pet.rg_name.id
  location  = var.resource_group_location
}

resource "random_string" "azurerm_search_service_name" {
  length   = 25
  upper    = false
  numeric  = false
  special  = false
}

resource "azurerm_search_service" "search" {
  name          = random_string.azurerm_search_service_name.result
  resource_group_name = azurerm_resource_group.rg.name
  location      = azurerm_resource_group.rg.location
  sku           = var.sku
  replica_count = var.replica_count
  partition_count = var.partition_count
}

```

3. Create a file named `outputs.tf` and insert the following code:

```

Terraform

output "resource_group_name" {
  value = azurerm_resource_group.rg.name
}

output "azurerm_search_service_name" {
  value = azurerm_search_service.search.name
}

```

4. Create a file named `providers.tf` and insert the following code:

```

Terraform

terraform {
  required_version = ">=1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~>3.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~>3.0"
    }
  }
}

```

```
provider "azurerm" {
  features {}
}
```

5. Create a file named `variables.tf` and insert the following code:

Terraform

```
variable "resource_group_location" {
  type     = string
  description = "Location for all resources."
  default    = "eastus"
}

variable "resource_group_name_prefix" {
  type     = string
  description = "Prefix of the resource group name that's combined with a
random ID so name is unique in your Azure subscription."
  default    = "rg"
}

variable "sku" {
  description = "The pricing tier of the search service you want to create
(for example, basic or standard)."
  default    = "standard"
  type       = string
  validation {
    condition    = contains(["free", "basic", "standard", "standard2",
"standard3", "storage_optimized_l1", "storage_optimized_l2"], var.sku)
    error_message = "The sku must be one of the following values: free,
basic, standard, standard2, standard3, storage_optimized_l1,
storage_optimized_l2."
  }
}

variable "replica_count" {
  type     = number
  description = "Replicas distribute search workloads across the service. You
need at least two replicas to support high availability of query workloads
(not applicable to the free tier)."
  default    = 1
  validation {
    condition    = var.replica_count >= 1 && var.replica_count <= 12
    error_message = "The replica_count must be between 1 and 12."
  }
}

variable "partition_count" {
  type     = number
  description = "Partitions allow for scaling of document count as well as
faster indexing by sharding your index over multiple search units."
  default    = 1
  validation {
```

```
    condition      = contains([1, 2, 3, 4, 6, 12], var.partition_count)
    error_message = "The partition_count must be one of the following values:
1, 2, 3, 4, 6, 12."
}
}
```

Initialize Terraform

Run [terraform init](#) to initialize the Terraform deployment. This command downloads the Azure provider required to manage your Azure resources.

Console

```
terraform init -upgrade
```

Key points:

- The `-upgrade` parameter upgrades the necessary provider plugins to the newest version that complies with the configuration's version constraints.

Create a Terraform execution plan

Run [terraform plan](#) to create an execution plan.

Console

```
terraform plan -out main.tfplan
```

Key points:

- The `terraform plan` command creates an execution plan, but doesn't execute it. Instead, it determines what actions are necessary to create the configuration specified in your configuration files. This pattern allows you to verify whether the execution plan matches your expectations before making any changes to actual resources.
- The optional `-out` parameter allows you to specify an output file for the plan. Using the `-out` parameter ensures that the plan you reviewed is exactly what is applied.

Apply a Terraform execution plan

Run [terraform apply](#) to apply the execution plan to your cloud infrastructure.

Console

```
terraform apply main.tfplan
```

Key points:

- The example `terraform apply` command assumes you previously ran `terraform plan -out main.tfplan`.
- If you specified a different filename for the `-out` parameter, use that same filename in the call to `terraform apply`.
- If you didn't use the `-out` parameter, call `terraform apply` without any parameters.

Verify the results

1. Get the Azure resource name in which the Azure AI Search service was created.

Console

```
resource_group_name=$(terraform output -raw resource_group_name)
```

2. Get the Azure AI Search service name.

Console

```
azurerm_search_service_name=$(terraform output -raw  
azurerm_search_service_name)
```

3. Run `az search service show` to show the Azure AI Search service you created in this article.

Azure CLI

```
az search service show --name $azurerm_search_service_name \  
--resource-group $resource_group_name
```

Clean up resources

When you no longer need the resources created via Terraform, do the following steps:

1. Run `terraform plan ↗` and specify the `destroy` flag.

Console

```
terraform plan -destroy -out main.destroy.tfplan
```

Key points:

- The `terraform plan` command creates an execution plan, but doesn't execute it. Instead, it determines what actions are necessary to create the configuration specified in your configuration files. This pattern allows you to verify whether the execution plan matches your expectations before making any changes to actual resources.
- The optional `-out` parameter allows you to specify an output file for the plan. Using the `-out` parameter ensures that the plan you reviewed is exactly what is applied.

2. Run [terraform apply](#) to apply the execution plan.

Console

```
terraform apply main.destroy.tfplan
```

Troubleshoot Terraform on Azure

[Troubleshoot common problems when using Terraform on Azure](#)

Next steps

[Create an Azure AI Search index using the Azure portal](#)

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

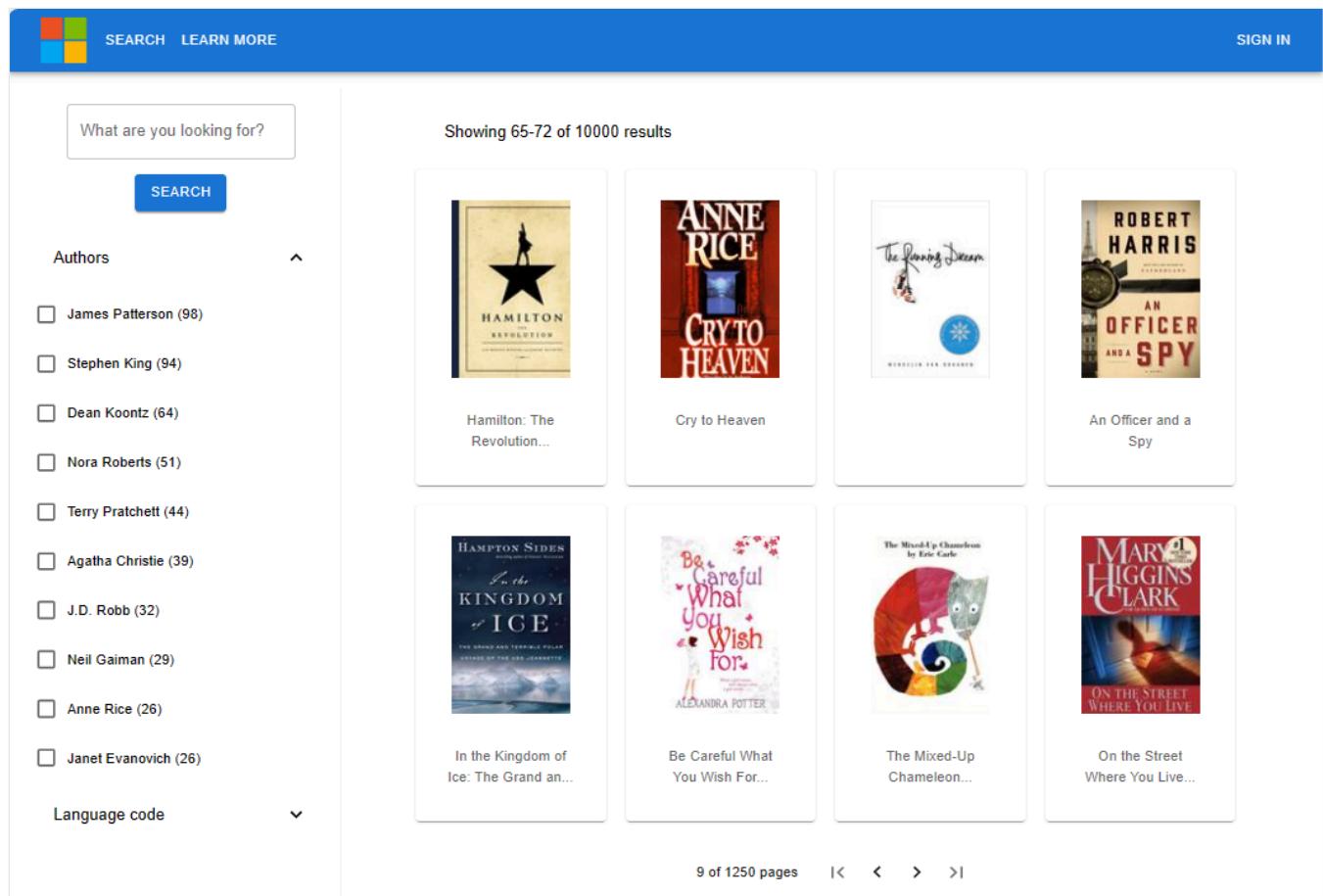
Step 1 - Overview of adding search to a static web app with .NET

09/23/2025

This tutorial builds a website that searches through a catalog of books and then deploys the website to an Azure Static Web App.

What does the sample do?

This sample website provides access to a catalog of 10,000 books. You can search the catalog by entering text in the search bar. While you enter text, the website uses the search index's suggestion feature to autocomplete the text. When the query finishes, the website displays the list of books with a portion of the details. You can select a book to see all the details, stored in the search index, of the book.



The screenshot shows a web application for searching a book catalog. At the top, there is a blue header bar with a Microsoft logo, a 'SEARCH' button, a 'LEARN MORE' link, and a 'SIGN IN' link. Below the header, a search bar contains the placeholder text 'What are you looking for?'. A blue 'SEARCH' button is positioned just below the search bar. To the left of the search results, there is a sidebar with a 'Authors' section containing a list of authors with their counts: James Patterson (98), Stephen King (94), Dean Koontz (64), Nora Roberts (51), Terry Pratchett (44), Agatha Christie (39), J.D. Robb (32), Neil Gaiman (29), Anne Rice (26), and Janet Evanovich (26). There is also a 'Language code' section with a dropdown arrow. The main content area displays a grid of book cards. The first row contains four cards: 'Hamilton: The Revolution...', 'Cry to Heaven', 'The Running Dream', and 'An Officer and a Spy'. The second row contains four cards: 'In the Kingdom of Ice: The Grand and Terrible Polar Voyage of the Jeannette', 'Be Careful What You Wish For...', 'The Mixed-Up Chameleon', and 'On the Street Where You Live...'. Each card includes the book title, author, and a small thumbnail image. At the bottom of the page, there is a footer with the text '9 of 1250 pages' and navigation icons for page navigation.

The search experience includes:

- **Search** – provides search functionality for the application.
- **Suggest** – provides suggestions as the user is typing in the search bar.
- **Facets and filters** - provides a faceted navigation structure that filters by author or language.

- [Paginated results](#) - provides paging controls for scrolling through results.
- [Document Lookup](#) – looks up a document by ID to retrieve all of its contents for the details page.

How is the sample organized?

The [sample code](#) includes the following components:

 Expand table

App	Purpose	GitHub Repository Location
client	React app (presentation layer) to display books, with search. It calls the Azure Function app.	/azure-search-static-web-app/client
api	Azure .NET Function app (business layer) - calls the Azure AI Search API using .NET SDK	/azure-search-static-web-app/api
bulk insert	.NET project to create the index and add documents to it.	/azure-search-static-web-app/bulk-insert

Set up your development environment

Create services and install the following software for your local development environment.

- [Azure AI Search](#), any region or tier
- [.NET 9](#) or latest version
- [Git](#)
- [Visual Studio Code](#)
- [C# Dev Tools extension for Visual Studio Code](#)
- [Azure Static Web App extension for Visual Studio Code](#)

This tutorial doesn't run the Azure Function API locally. If you want to run it locally, install [azure-functions-core-tools](#).

Fork and clone the search sample with git

To deploy the Static Web App, you need to fork the sample repository. The web apps use your GitHub fork location to decide the build actions and deployment content. Code execution in the Static Web App happens remotely, with Azure Static Web Apps reading the code from your forked sample.

1. On GitHub, fork the [azure-search-static-web-app repository](#).

Complete the [fork process](#) in your web browser with your GitHub account. This tutorial uses your fork as part of the deployment to an Azure Static Web App.

2. At a Bash terminal, download your forked sample application to your local computer.

Replace `YOUR-GITHUB-ALIAS` with your GitHub alias.

Bash

```
git clone https://github.com/YOUR-GITHUB-ALIAS/azure-search-static-web-app.git
```

3. At the same Bash terminal, go into your forked repository for this website search example:

Bash

```
cd azure-search-static-web-app
```

4. Use the Visual Studio Code command, `code .` to open your forked repository. You accomplish the remaining tasks from Visual Studio Code, unless specified.

Bash

```
code .
```

Next steps

- [Create an index and load it with documents](#)
- [Deploy your Static Web App](#)

Step 2 - Create and load the search index

Continue to build your search-enabled website by following these steps:

- Create a new index
- Load data

The program uses [Azure.Search.Documents](#) in the Azure SDK for .NET:

- NuGet package [Azure.Search.Documents](#)
- Reference Documentation

Before you start, make sure you have room on your search service for a new index. The free tier limit is three indexes. The Basic tier limit is 15.

Prepare the bulk import script for Search

1. In Visual Studio Code, open the `Program.cs` file in the subdirectory, `azure-search-static-web-app/bulk-insert`, replace the following variables with your own values to authenticate with the Azure Search SDK.

- YOUR-SEARCH-SERVICE-NAME (not the full URL)
- YOUR-SEARCH-ADMIN-API-KEY (see [Find API keys](#))

C#

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using AzureSearch.BulkInsert;
using ServiceStack;

const string BOOKS_URL = "https://raw.githubusercontent.com/Azure-
Samples/azure-search-sample-data/main/good-books/books.csv";
const string SEARCH_ENDPOINT = "https://YOUR-SEARCH-RESOURCE-
NAME.search.windows.net";
const string SEARCH_KEY = "YOUR-SEARCH-ADMIN-KEY";
const string SEARCH_INDEX_NAME = "good-books";

Uri searchEndpointUri = new(SEARCH_ENDPOINT);

SearchClient client = new(
    searchEndpointUri,
    SEARCH_INDEX_NAME,
    new AzureKeyCredential(SEARCH_KEY));
```

```

SearchIndexClient clientIndex = new(
    searchEndpointUri,
    new AzureKeyCredential(SEARCH_KEY));

await CreateIndexAsync(clientIndex);
await BulkInsertAsync(client);

static async Task CreateIndexAsync(SearchIndexClient clientIndex)
{
    Console.WriteLine("Creating (or updating) search index");
    SearchIndex index = new BookSearchIndex(SEARCH_INDEX_NAME);
    var result = await clientIndex.CreateOrUpdateIndexAsync(index);

    Console.WriteLine(result);
}

static async Task BulkInsertAsync(SearchClient client)
{
    Console.WriteLine("Download data file");
    using HttpClient httpClient = new();

    var csv = await httpClient.GetStringAsync(BOOKS_URL);

    Console.WriteLine("Reading and parsing raw CSV data");
    var books =
        csv.ReplaceFirst("book_id", "id").FromCsv<List<BookModel>>();

    Console.WriteLine("Uploading bulk book data");
    _ = await client.UploadDocumentsAsync(books);

    Console.WriteLine("Finished bulk inserting book data");
}

```

2. Open an integrated terminal in Visual Studio Code for the project directory's subdirectory, `azure-search-static-web-app/bulk-insert`.

3. Run the following command to install the dependencies.

Bash

```
dotnet restore
```

Run the bulk import script for Search

1. Still in the same subdirectory (`azure-search-static-web-app/bulk-insert`), run the program:

Bash

```
dotnet run
```

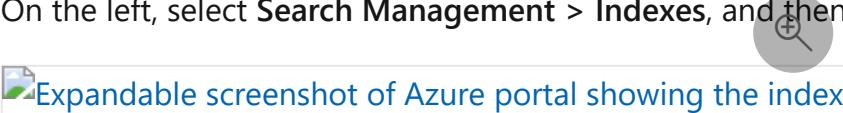
- As the code runs, the console displays progress. You should see the following output.

Bash

```
Creating (or updating) search index
Status: 201, Value: Azure.Search.Documents.Indexes.Models.SearchIndex
Download data file
Reading and parsing raw CSV data
Uploading bulk book data
Finished bulk inserting book data
```

Review the new search index

Once the upload completes, the search index is ready to use. Review your new index in Azure portal.

- In Azure portal, [find your search service ↗](#).
- On the left, select **Search Management > Indexes**, and then select the good-books index.

- By default, the index opens in the **Search Explorer** tab. Select **Search** to return documents from the index.


Rollback bulk import file changes

Use the following git command in the Visual Studio Code integrated terminal at the `bulk-insert` directory to roll back the changes to the `Program.cs` file. They aren't needed to continue the tutorial and you shouldn't save or push your API keys or search service name to your repo.

```
git
```

```
git checkout .
```

Next steps

[Deploy your Static Web App](#)

Last updated on 11/21/2025

Step 3 - Deploy the search-enabled .NET website

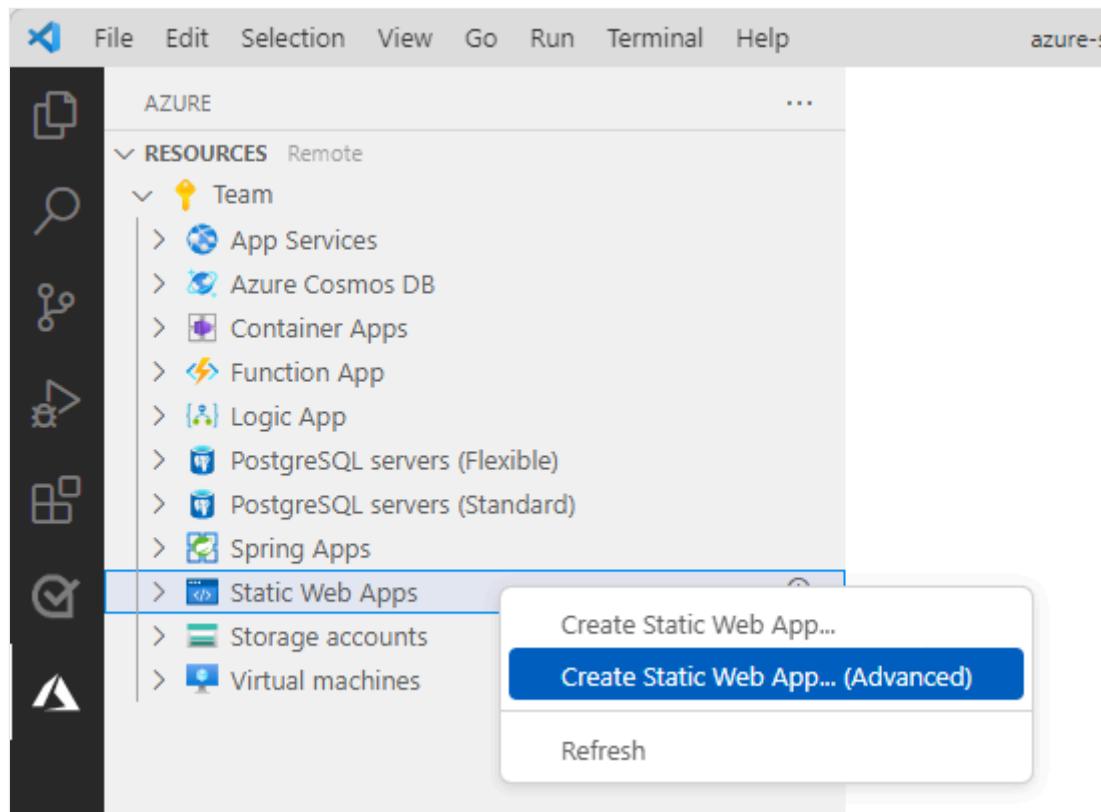
09/23/2025

Deploy the search-enabled website as an Azure Static Web Apps site. This deployment includes both the React app for the web pages, and the Function app for search operations.

The static web app pulls the information and files for deployment from GitHub using your fork of the `azure-search-static-web-app` repository.

Create a Static Web App in Visual Studio Code

1. In Visual Studio Code, make sure you're at the repository root, and not the bulk-insert folder (for example, `azure-search-static-web-app`).
2. Select **Azure** from the Activity Bar, then open **Resources** from the side bar.
3. Right-click **Static Web Apps** and then select **Create Static Web App (Advanced)**. If you don't see this option, verify that you have the Azure Functions extension for Visual Studio Code.



4. If you see a pop-up window asking you to commit your changes, don't do this. The secrets from the bulk import step shouldn't be committed to the repository.

To roll back the changes, in Visual Studio Code select the Source Control icon in the Activity bar, then select each changed file in the Changes list and select the **Discard changes** icon.

5. Follow the prompts to create the static web app:

 Expand table

Prompt	Enter
Select a resource group for new resources.	Create a new resource group for the static app.
Enter the name for the new Static Web App.	Give your static app a name, such as <code>my-demo-static-web-app</code> .
Select a SKU	Select the free SKU for this tutorial.
Select a location for new resources.	Choose a region near you.
Choose build preset to configure default project structure.	Select Custom .
Select the location of your client application code	<code>client</code> This is the path, from the root of the repository, to your static web app.
Enter the path of your build output...	<code>build</code> This is the path, from your static web app, to your generated files.

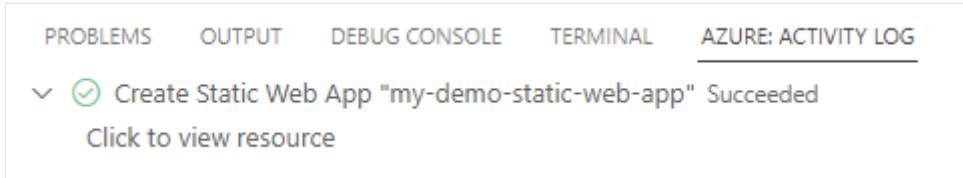
If you get an error about an incorrect region, make sure the resource group and static web app resource are in one of the supported regions listed in the error response.

6. When the static web app is created, a GitHub workflow YML file is also created locally and on GitHub in your fork. This workflow executes in your fork, building and deploying the static web app and functions.

Check the status of static web app deployment using any of these approaches:

- Select **Open Actions in GitHub** from the Notifications. This opens a browser window pointed to your forked repo.
- Select the **Actions** tab in your forked repository. You should see a list of all workflows on your fork.

- Select the **Azure: Activity Log** in Visual Code. You should see a message similar to the following screenshot.



The screenshot shows the Visual Studio Code interface with the "AZURE: ACTIVITY LOG" tab selected in the top navigation bar. Below the tab, there is a single log entry: "Create Static Web App "my-demo-static-web-app"" followed by the status "Succeeded". A link "Click to view resource" is provided below the log entry.

Get the Azure AI Search query key in Visual Studio Code

While you might be tempted to reuse your search admin key for query purposes that isn't following the principle of least privilege. The Azure Function should use the query key to conform to least privilege.

1. In Visual Studio Code, open a new terminal window.
2. Get the query API key with this Azure CLI command:

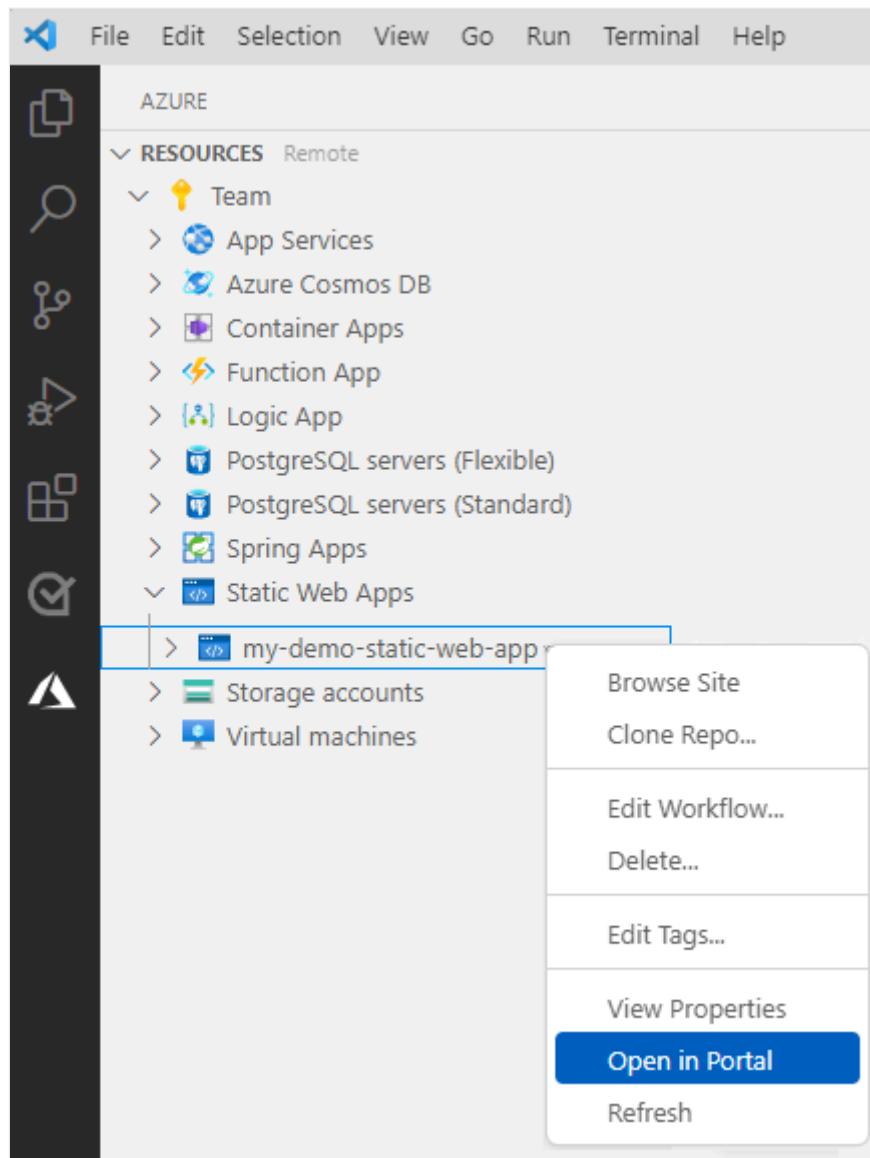
```
Azure CLI  
  
az search query-key list --resource-group YOUR-SEARCH-SERVICE-RESOURCE-GROUP  
--service-name YOUR-SEARCH-SERVICE-NAME
```

3. Keep this query key to use in the next section. The query key authorizes read access to a search index.

Add environment variables in Azure portal

The Azure Function app won't return search data until the search secrets are in settings.

1. Select **Azure** from the Activity Bar.
2. Right-click on your Static Web Apps resource then select **Open in Portal**.



3. Select **Environment variables** then select + Add application setting.

A screenshot of the Azure portal showing the configuration for the 'my-demo-static-web-app'. The left sidebar has links for Home, my-demo-static-web-app, Overview, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Settings, Configuration, Environment variables (which is selected and highlighted in grey), Application Insights, and Custom domains. The main content area shows the 'my-demo-static-web-app | Environment variables' page. It includes sections for Overview, Environment variables, and Application settings. The 'Add application setting' button is highlighted with a red box. A table lists environment variables: Name (SearchApiKey) and Value (Hidden value. Click to show value).

4. Add each of the following settings:

Expand table

Setting	Your Search resource value
SearchApiKey	Your search query key
SearchServiceName	Your search resource name
SearchIndexName	good-books
SearchFacets	authors*,language_code

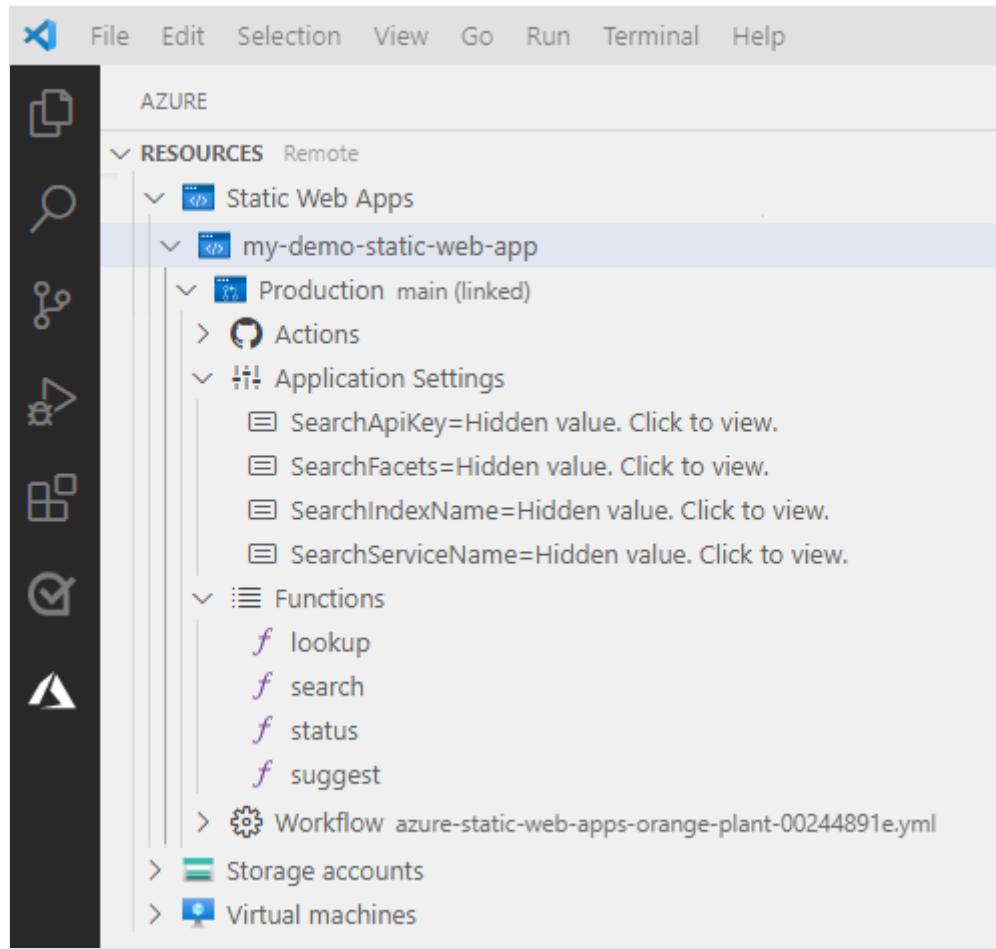
Azure AI Search requires different syntax for filtering collections than it does for strings. Add a `*` after a field name to denote that the field is of type `Collection(Edm.String)`. This allows the Azure Function to add filters correctly to queries.

5. Check your settings to make sure they look like the following screenshot.

Name	Value	Delete
SearchApiKey	Hidden value. Click to show value	
SearchFacets	Hidden value. Click to show value	
SearchIndexName	Hidden value. Click to show value	
SearchServiceName	Hidden value. Click to show value	

6. Return to Visual Studio Code.

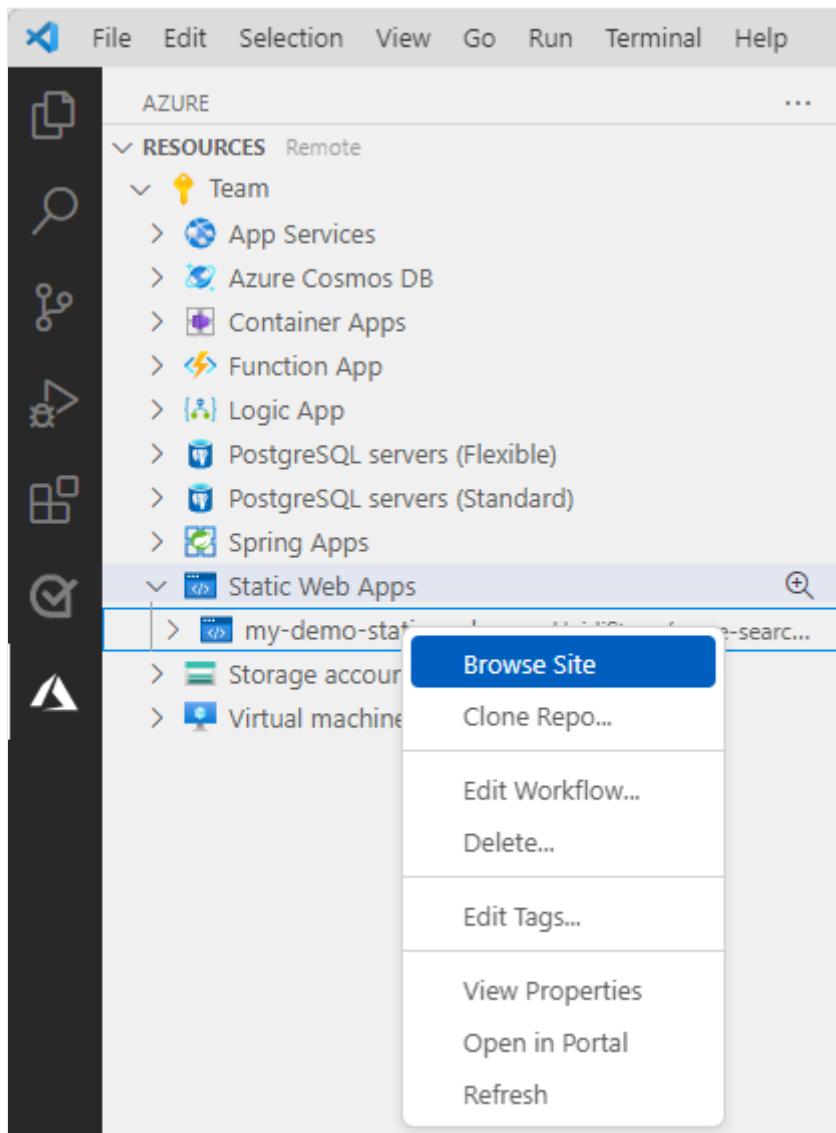
7. Refresh your static web app to see the application settings and functions.



If you don't see the application settings, revisit the steps for updating and relaunching the GitHub workflow.

Use search in your static web app

1. In Visual Studio Code, open the [Activity bar](#), and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Static Web Apps` area and find the static web app you created for this tutorial.
3. Right-click the static web app name and select **Browse site**.



4. Select **Open** in the pop-up dialog.
5. In the website search bar, enter a search query such as `code`, so the suggest feature suggests book titles. Select a suggestion or continue entering your own query. Press enter when you've completed your search query.
6. Review the results then select one of the books to see more details.

Troubleshooting

If the web app didn't deploy or work, use the following list to determine and fix the issue:

- **Did the deployment succeed?**

In order to determine if your deployment succeeded, you need to go to *your* fork of the sample repo and review the success or failure of the GitHub action. There should be only one action and it should have static web app settings for the `app_location`, `api_location`,

and `output_location`. If the action didn't deploy successfully, dive into the action logs and look for the last failure.

- **Does the client (front-end) application work?**

You should be able to get to your web app and it should successfully display. If the deployment succeeded but the website doesn't display, this might be an issue with how the static web app is configured for rebuilding the app, once it is on Azure.

- **Does the API (serverless back-end) application work?**

You should be able to interact with the client app, searching for books and filtering. If the form doesn't return any values, open the browser's developer tools, and determine if the HTTP calls to the API were successful. If the calls weren't successful, the most likely reason if the static web app configurations for the API endpoint name and search query key are incorrect.

If the path to the Azure function code (`api_location`) isn't correct in the YML file, the application loads but won't call any of the functions that provide integration with Azure AI Search. Revisit the deployment section to make sure paths are correct.

Clean up resources

To clean up the resources created in this tutorial, delete the resource group or individual resources.

1. In Visual Studio Code, open the [Activity bar](#), and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Static Web Apps` area and find the app you created for this tutorial.
3. Right-click the app name then select **Delete**.
4. If you no longer want the GitHub fork of the sample, remember to delete that on GitHub. Go to your fork's **Settings** then delete the repository.
5. To delete Azure AI Search, [find your search service](#) and select **Delete** at the top of the page.

Next steps

- [Understand Search integration for the search-enabled website](#)

Step 4 - Explore the .NET search code

09/23/2025

In the previous lessons, you added search to a static web app. This lesson highlights the essential steps that establish integration. If you're looking for a cheat sheet on how to integrate search into your web app, this article explains what you need to know.

Azure SDK Azure.Search.Documents

The Function app uses the Azure SDK for Azure AI Search:

- NuGet: [Azure.Search.Documents](#)
- Reference Documentation: [Client Library](#)

The function app authenticates through the SDK to the cloud-based Azure AI Search API using your resource name, resource key, and index name. The secrets are stored in the static web app settings and pulled in to the function as environment variables.

Configure secrets in a local.settings.json file

```
JSON

{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",
    "SearchApiKey": "",
    "SearchServiceName": "",
    "SearchIndexName": "good-books"
  },
  "Host": {
    "CORS": "*"
  }
}
```

Azure Function: Search the catalog

The [Search API](#) takes a search term and searches across the documents in the search index, returning a list of matches. Through the Suggest API, partial strings are sent to the search engine as the user types, suggesting search terms such as book titles and authors across the documents in the search index, and returning a small list of matches.

The Azure function pulls in the search configuration information, and fulfills the query.

The search suggester, `sg`, is defined in the [schema file](#) used during bulk upload.

C#

```
using Azure;
using Azure.Core.Serialization;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;
using System.Net;
using System.Text.Json;
using System.Text.Json.Serialization;
using WebSearch.Models;
using SearchFilter = WebSearch.Models.SearchFilter;

namespace WebSearch.Function
{
    public class Search
    {
        private static string searchApiKey =
Environment.GetEnvironmentVariable("SearchApiKey",
EnvironmentVariableTarget.Process);
        private static string searchServiceName =
Environment.GetEnvironmentVariable("SearchServiceName",
EnvironmentVariableTarget.Process);
        private static string searchIndexName =
Environment.GetEnvironmentVariable("SearchIndexName",
EnvironmentVariableTarget.Process) ?? "good-books";

        private readonly ILogger<Lookup> _logger;

        public Search	ILogger<Lookup> logger)
        {
            _logger = logger;
        }

        [Function("search")]
        public async Task<HttpResponseData> RunAsync(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post")] HttpRequestData
req,
            FunctionContext executionContext)
        {
            string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
            var data = JsonSerializer.Deserialize<RequestBodySearch>(requestBody);

            // Azure AI Search
            Uri serviceEndpoint =
new($"https://{{searchServiceName}}.search.windows.net/");
            ...
        }
    }
}
```

```

        SearchClient searchClient = new(
            serviceEndpoint,
            searchIndexName,
            new AzureKeyCredential(searchApiKey)
        );

        SearchOptions options = new()

        {
            Size = data.Size,
            Skip = data.Skip,
            IncludeTotalCount = true,
            Filter = CreateFilterExpression(data.Filters)
        };
        options.Facets.Add("authors");
        options.Facets.Add("language_code");

        SearchResults<SearchDocument> searchResults =
searchClient.Search<SearchDocument>(data.SearchText, options);

        var facetOutput = new Dictionary<string, IList<FacetValue>>();
        foreach (var facetResult in searchResults.Facets)
        {
            facetOutput[facetResult.Key] = facetResult.Value
                .Select(x => new FacetValue { value =
x.Value.ToString(), count = x.Count })

                .ToList();
        }

        // Data to return
        var output = new SearchOutput
        {
            Count = searchResults.TotalCount,
            Results = searchResults.GetResults().ToList(),
            Facets = facetOutput
        };

        var response = req.CreateResponse(HttpStatusCode.Found);

        // Serialize data
        var serializer = new JsonObjectSerializer(
            new JsonSerializerOptions(JsonSerializerDefaults.Web));
        await response.WriteAsJsonAsync(output, serializer);

        return response;
    }

    public static string CreateFilterExpression(List<SearchFilter> filters)
    {
        if (filters is null or { Count: <= 0 })
        {
            return null;
        }
    }
}

```

```

        List<string> filterExpressions = new();

        List<SearchFilter> authorFilters = filters.Where(f => f.field == "authors").ToList();
        List<SearchFilter> languageFilters = filters.Where(f => f.field == "language_code").ToList();

        List<string> authorFilterValues = authorFilters.Select(f => f.value).ToList();

        if (authorFilterValues.Count > 0)
        {
            string filterStr = string.Join(", ", authorFilterValues);
            filterExpressions.Add($"\"authors\"/any(t: search.in(t, '{filterStr}', ','))");
        }

        List<string> languageFilterValues = languageFilters.Select(f => f.value).ToList();
        foreach (var value in languageFilterValues)
        {
            filterExpressions.Add($"language_code eq '{value}'");
        }

        return string.Join(" and ", filterExpressions);
    }
}
}

```

Client: Search from the catalog

Call the Azure Function in the React client at `\client\src\pages\Search\Search.jsx` with the following code to search for books.

```

C#

import React, { useEffect, useState, Suspense } from 'react';
import fetchInstance from '..../url-fetch';
import CircularProgress from '@mui/material/CircularProgress';
import { useLocation, useNavigate } from "react-router-dom";

import Results from '..../components/Results/Results';
import Pager from '..../components/Pager/Pager';
import Facets from '..../components/Facets/Facets';
import SearchBar from '..../components/SearchBar/SearchBar';

import './Search.css';

export default function Search() {

```

```
let location = useLocation();
const navigate = useNavigate();

const [results, setResults] = useState([]);
const [resultCount, setResultCount] = useState(0);
const [currentPage, setCurrentPage] = useState(1);
const [q, setQ] = useState(new URLSearchParams(location.search).get('q') ?? "*");
const [top] = useState(new URLSearchParams(location.search).get('top') ?? 8);
const [skip, setSkip] = useState(new URLSearchParams(location.search).get('skip') ?? 0);
const [filters, setFilters] = useState([]);
const [facets, setFacets] = useState({});
const [isLoading, setIsLoading] = useState(true);

let resultsPerPage = top;

// Handle page changes in a controlled manner
function handlePageChange(newPage) {
  setCurrentPage(newPage);
}

// Calculate skip value and fetch results when relevant parameters change
useEffect(() => {
  // Calculate skip based on current page
  const calculatedSkip = (currentPage - 1) * top;

  // Only update if skip has actually changed
  if (calculatedSkip !== skip) {
    setSkip(calculatedSkip);
    return; // Skip the fetch since skip will change and trigger another
  }
  useEffect(() => {
    // Proceed with fetch
    setIsLoading(true);

    const body = {
      q,
      top,
      skip,
      filters
    };

    fetchInstance('/api/search', { body, method: 'POST' })
      .then(response => {
        setResults(response.results);
        setFacets(response.facets);
        setResultCount(response.count);
        setIsLoading(false);
      })
      .catch(error => {
        console.log(error);
        setIsLoading(false);
      })
  }, [skip]);
}

handlePageChange(2);
```

```

        });
    }, [q, top, skip, filters, currentPage]);

    // pushing the new search term to history when q is updated
    // allows the back button to work as expected when coming back from the details
page
    useEffect(() => {
        navigate('/search?q=' + q);
        setCurrentPage(1);
        setFilters([]);
        // eslint-disable-next-line react-hooks/exhaustive-deps
    }, [q]);

let postSearchHandler = (searchTerm) => {
    setQ(searchTerm);
}

// filters should be applied across entire result set,
// not just within the current page
const updateFilterHandler = (newFilters) => {

    // Reset paging
    setSkip(0);
    setCurrentPage(1);

    // Set filters
    setFilters(newFilters);
};

return (
    <main className="main main--search container-fluid">
        <div className="row">
            <div className="search-bar-column col-md-3">
                <div className="search-bar-column-container">
                    <SearchBar postSearchHandler={postSearchHandler} query={q} width={false}></SearchBar>
                </div>
                <Facets facets={facets} filters={filters} setFilters={updateFilterHandler}></Facets>
            </div>
            <div className="search-bar-results">
                {isLoading ? (
                    <div className="col-md-9">
                        <CircularProgress />
                    </div>
                ) : (
                    <div className="search-results-container">
                        <Results documents={results} top={top} skip={skip} count={resultCount} query={q}></Results>
                        <Pager className="pager-style" currentPage={currentPage} resultCount={resultCount} resultsPerPage={resultsPerPage} onPageChange={handlePageChange}></Pager>
                    </div>
                )}
            </div>
        </div>
    </main>
)

```

```
        )}
      </div>
    </div>
  </main>
);
}
```

Client: Suggestions from the catalog

The Suggest function API is called in the React app at

`\client\src\components\SearchBar\SearchBar.jsx` as part of the [Material UI's Autocomplete component](#). This component uses the input text to search for authors and books that match the input text then displays those possible matches at selectable items in the drop-down list.

C#

```
import React, { useState, useEffect } from 'react';
import { TextField, Autocomplete, Button, Box } from '@mui/material';
import fetchInstance from '../.../url-fetch';
import './SearchBar.css';

export default function SearchBar({ postSearchHandler, query, width }) {
  const [q, setQ] = useState(() => query || '');
  const [suggestions, setSuggestions] = useState([]);

  const search = (value) => {
    postSearchHandler(value);
  };

  useEffect(() => {
    if (q) {

      const body = { q, top: 5, suggester: 'sg' };

      fetchInstance('/api/suggest', { body, method: 'POST' })
        .then(response => {
          setSuggestions(response.suggestions.map(s => s.text));
        })
        .catch(error => {
          console.log(error);
          setSuggestions([]);
        });
    }
  }, [q]);

  const onInputChangeHandler = (event, value) => {
    setQ(value);
  };
}
```

```
const onChangeHandler = (event, value) => {
  setQ(value);
  search(value);
};

const onEnterButton = (event) => {
  // if enter key is pressed
  if (event.key === 'Enter') {
    search(q);
  }
};

return (
  <div
    className={width ? "search-bar search-bar-wide" : "search-bar search-bar-narrow"}>
    <Box className="search-bar-box">
      <Autocomplete
        className="autocomplete"
        freeSolo
        value={q}
        options={suggestions}
        onInputChange={onInputChangeHandler}
        onChange={onChangeHandler}
        disableClearable
        renderInput={(params) => (
          <TextField
            {...params}
            id="search-box"
            className="form-control rounded-0"
            placeholder="What are you looking for?"
            onBlur={() => setSuggestions([])}
            onClick={() => setSuggestions([])}
          />
        )}
      />
      <div className="search-button" >
        <Button variant="contained" color="primary" onClick={() => {
          search(q)
        }}>
          Search
        </Button>
      </div>
    </Box>
  </div>
);
}
```

Azure Function: Get specific document

The [Document Lookup API](#) takes an ID and returns the document object from the Search Index.

C#

```
using Azure;
using Azure.Core.Serialization;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;
using System.Net;
using System.Text.Json;
using WebSearch.Models;

namespace WebSearch.Function
{
    public class Lookup
    {
        private static string searchApiKey =
Environment.GetEnvironmentVariable("SearchApiKey",
EnvironmentVariableTarget.Process);
        private static string searchServiceName =
Environment.GetEnvironmentVariable("SearchServiceName",
EnvironmentVariableTarget.Process);
        private static string searchIndexName =
Environment.GetEnvironmentVariable("SearchIndexName",
EnvironmentVariableTarget.Process) ?? "good-books";

        private readonly ILogger<Lookup> _logger;

        public Lookup(ILogger<Lookup> logger)
        {
            _logger = logger;
        }

        [Function("lookup")]
        public async Task<HttpResponseData> RunAsync(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")]
HttpRequestData req,
            FunctionContext executionContext)
        {

            // Get Document Id
            var query = System.Web.HttpUtility.ParseQueryString(req.Url.Query);
            string documentId = query[ "id" ].ToString();

            // Azure AI Search
            Uri serviceEndpoint =
```

```

new($"https://{{searchServiceName}}.search.windows.net/");

    SearchClient searchClient = new(
        serviceEndpoint,
        searchIndexName,
        new AzureKeyCredential(searchApiKey)
    );

    var getDocumentResponse = await
searchClient.GetDocumentAsync<SearchDocument>(documentId);

    // Data to return
    var output = new LookupOutput
    {
        Document = getDocumentResponse.Value
    };

    var response = req.CreateResponse(HttpStatusCode.Found);

    // Serialize data
    var serializer = new JsonObjectSerializer(
        new JsonSerializerOptions(JsonSerializerDefaults.Web));
    await response.WriteAsJsonAsync(output, serializer);

    return response;
}
}
}
}

```

Client: Get specific document

This function API is called in the React app at `\client\src\pages\Details\Details.jsx` as part of component initialization:

```

C#

import React, { useState, useEffect } from "react";
import { useParams } from 'react-router-dom';
import Rating from '@mui/material/Rating';
import CircularProgress from '@mui/material/CircularProgress';
import Tabs from '@mui/material/Tabs';
import Tab from '@mui/material/Tab';
import Box from '@mui/material/Box';

import fetchInstance from '.../url-fetch';

import "./Details.css";

function CustomTabPanel(props) {

```

```
const { children, value, index, ...other } = props;

return (
  <div
    className="tab-panel"
    role="tabpanel"
    hidden={value !== index}
    id={`simple-tabpanel-${index}`}
    aria-labelledby={`simple-tab-${index}`}
    {...other}
    // Ensure it takes full width
  >
  {value === index && <Box className="tab-panel-value">{children}</Box>}
</div>
);
}

export default function BasicTabs() {
  const { id } = useParams();
  const [document, setDocument] = useState({});
  const [value, setValue] = React.useState(0);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setIsLoading(true);
    fetchInstance('/api/lookup', { query: { id } })
      .then(response => {
        console.log(JSON.stringify(response))
        const doc = response.document;
        setDocument(doc);
        setIsLoading(false);
      })
      .catch(error => {
        console.log(error);
        setIsLoading(false);
      });
  }, [id]);

  const handleChange = (event, newValue) => {
    setValue(newValue);
  };

  if (isLoading || !id || Object.keys(document).length === 0) {
    return (
      <div className="loading-container">
        <CircularProgress />
        <p>Loading...</p>
      </div>
    );
  }

  return (
    <Box className="details-box-parent">
```

```

<Box className="details-tab-box-header">
  <Tabs value={value} onChange={handleChange} aria-label="book-details-tabs">
    <Tab label="Result" />
    <Tab label="Raw Data" />
  </Tabs>
</Box>
<CustomTabPanel value={value} index={0} className="tab-panel box-content">
  <div className="card-body">
    <h5 className="card-title">{document.original_title}</h5>
    <img className="image" src={document.image_url} alt="Book cover"></img>
    <p className="card-text">{document.authors?.join(' ') - document.original_publication_year}</p>
    <p className="card-text">ISBN {document.isbn}</p>
    <Rating name="half-rating-read" value={parseInt(document.average_rating)} precision={0.1} readOnly></Rating>
    <p className="card-text">{document.ratings_count} Ratings</p>
  </div>
</CustomTabPanel>
<CustomTabPanel value={value} index={1} className="tab-panel">
  <div className="card-body text-left card-text details-custom-tab-panel-json-div" >
    <pre><code>
      {JSON.stringify(document, null, 2)}
    </code></pre>
  </div>
</CustomTabPanel>
</Box>
);
}

```

C# models to support function app

The following models are used to support the functions in this app.

```

C#

using Azure.Search.Documents.Models;
using System.Text.Json.Serialization;

namespace WebSearch.Models
{
  public class RequestBodyLookUp
  {
    [JsonPropertyName("id")]
    public string Id { get; set; }
  }

  public class RequestBodySuggest
  {
    [JsonPropertyName("q")]
    public string SearchText { get; set; }
  }
}

```

```
[JsonPropertyName("top")]
public int Size { get; set; }

[JsonPropertyName("suggester")]
public string SuggesterName { get; set; }
}

public class RequestBodySearch
{
    [JsonPropertyName("q")]
    public string SearchText { get; set; }

    [JsonPropertyName("skip")]
    public int Skip { get; set; }

    [JsonPropertyName("top")]
    public int Size { get; set; }

    [JsonPropertyName("filters")]
    public List<SearchFilter> Filters { get; set; }
}

public class SearchFilter
{
    public string field { get; set; }
    public string value { get; set; }
}

public class FacetValue
{
    public string value { get; set; }
    public long? count { get; set; }
}

class SearchOutput
{
    [JsonPropertyName("count")]
    public long? Count { get; set; }
    [JsonPropertyName("results")]
    public List<SearchResult<SearchDocument>> Results { get; set; }
    [JsonPropertyName("facets")]
    public Dictionary<String, IList<FacetValue>> Facets { get; set; }
}
class LookupOutput
{
    [JsonPropertyName("document")]
    public SearchDocument Document { get; set; }
}
public class BookModel
{
    public string id { get; set; }
    public decimal? goodreads_book_id { get; set; }
    public decimal? best_book_id { get; set; }
    public decimal? work_id { get; set; }
}
```

```
public decimal? books_count { get; set; }
public string isbn { get; set; }
public string isbn13 { get; set; }
public string[] authors { get; set; }
public decimal? original_publication_year { get; set; }
public string original_title { get; set; }
public string title { get; set; }
public string language_code { get; set; }
public double? average_rating { get; set; }
public decimal? ratings_count { get; set; }
public decimal? work_ratings_count { get; set; }
public decimal? work_text_reviews_count { get; set; }
public decimal? ratings_1 { get; set; }
public decimal? ratings_2 { get; set; }
public decimal? ratings_3 { get; set; }
public decimal? ratings_4 { get; set; }
public decimal? ratings_5 { get; set; }
public string image_url { get; set; }
public string small_image_url { get; set; }
}
}
```

Next steps

To continue learning more about Azure AI Search development, try this next tutorial about indexing:

- [Index Azure SQL data](#)

Tutorial: Optimize indexing using the push API

Azure AI Search supports two basic methods for [importing data](#) into a search index: *pushing* your data into the index programmatically or *pulling* your data by pointing an [indexer](#) to a supported data source.

This tutorial explains how to efficiently index data using the [push model](#) by batching requests and using an exponential backoff retry strategy. You can download and run the [sample application](#). This tutorial also explains the key aspects of the application and what factors to consider when indexing data.

In this tutorial, you use C# and the [Azure.Search.Documents library](#) from the Azure SDK for .NET to:

- ✓ Create an index
- ✓ Test various batch sizes to determine the most efficient size
- ✓ Index batches asynchronously
- ✓ Use multiple threads to increase indexing speeds
- ✓ Use an exponential backoff retry strategy to retry failed documents

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Visual Studio](#).

Download files

Source code for this tutorial is in the [optimize-data-indexing/v11](#) folder in the [Azure-Samples/azure-search-dotnet-scale](#) GitHub repository.

Key considerations

The following factors affect indexing speeds. For more information, see [Index large data sets](#).

- **Pricing tier and number of partitions(replicas)**: Adding partitions or upgrading your tier increases indexing speeds.
- **Index schema complexity**: Adding fields and field properties lowers indexing speeds. Smaller indexes are faster to index.
- **Batch size**: The optimal batch size varies based on your index schema and dataset.

- **Number of threads/workers:** A single thread doesn't take full advantage of indexing speeds.
- **Retry strategy:** An exponential backoff retry strategy is a best practice for optimum indexing.
- **Network data transfer speeds:** Data transfer speeds can be a limiting factor. Index data from within your Azure environment to increase data transfer speeds.

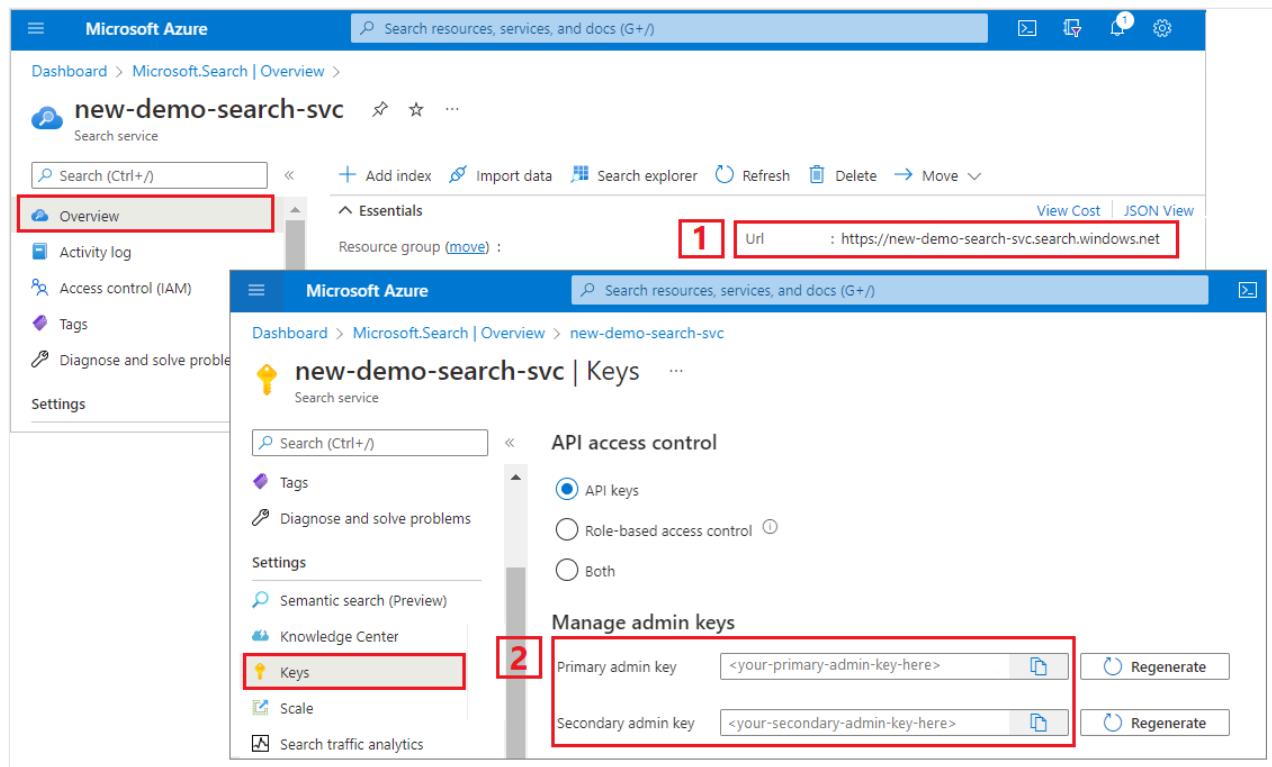
Create a search service

This tutorial requires an Azure AI Search service, which you can [create in the Azure portal](#). You can also [find an existing service](#) in your current subscription. To accurately test and optimize indexing speeds, we recommend using the same pricing tier you plan to use in production.

Get an admin key and URL for Azure AI Search

This tutorial uses key-based authentication. Copy an admin API key to paste into the `appsettings.json` file.

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview** and copy the endpoint. It should be in this format:
`https://my-service.search.windows.net`
3. From the left pane, select **Settings > Keys** and copy an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests to add, modify, or delete objects.



Set up your environment

1. Open the `OptimizeDataIndexing.sln` file in Visual Studio.
2. In Solution Explorer, edit the `appsettings.json` file with the connection information you collected in the previous step.

```
JSON
{
  "SearchServiceUri": "https://{{service-name}}.search.windows.net",
  "SearchServiceAdminApiKey": "",
  "SearchIndexName": "optimize-indexing"
}
```

Explore the code

After you update `appsettings.json`, the sample program in `OptimizeDataIndexing.sln` should be ready to build and run.

This code is derived from the C# section of [Quickstart: Full-text search](#), which provides detailed information about the basics of working with the .NET SDK.

This simple C#/NET console app performs the following tasks:

- Creates a new index based on the data structure of the C# `Hotel` class (which also references the `Address` class)
- Tests various batch sizes to determine the most efficient size
- Indexes data asynchronously
 - Using multiple threads to increase indexing speeds
 - Using an exponential backoff retry strategy to retry failed items

Before you run the program, take a minute to study the code and the index definitions for this sample. The relevant code is in several files:

- `Hotel.cs` and `Address.cs` contain the schema that defines the index
- `DataGenerator.cs` contains a simple class to make it easy to create large amounts of hotel data
- `ExponentialBackoff.cs` contains code to optimize the indexing process as described in this article
- `Program.cs` contains functions that create and delete the Azure AI Search index, indexes batches of data, and tests different batch sizes

Create the index

This sample program uses the Azure SDK for .NET to define and create an Azure AI Search index. It takes advantage of the `FieldBuilder` class to generate an index structure from a C# data model class.

The data model is defined by the `Hotel` class, which also contains references to the `Address` class. `FieldBuilder` drills down through multiple class definitions to generate a complex data structure for the index. Metadata tags are used to define the attributes of each field, such as whether it's searchable or sortable.

The following snippets from the `Hotel.cs` file specify a single field and a reference to another data model class.

```
C#
...
[SearchableField(IsSortable = true)]
public string HotelName { get; set; }
...
public Address Address { get; set; }
...
```

In the `Program.cs` file, the index is defined with a name and a field collection generated by the `FieldBuilder.Build(typeof(Hotel))` method, and then created as follows:

C#

```
private static async Task CreateIndexAsync(string indexName, SearchIndexClient indexClient)
{
    // Create a new search index structure that matches the properties of the Hotel
    // class.
    // The Address class is referenced from the Hotel class. The FieldBuilder
    // will enumerate these to create a complex data structure for the index.
    FieldBuilder builder = new FieldBuilder();
    var definition = new SearchIndex(indexName, builder.Build(typeof(Hotel)));

    await indexClient.CreateIndexAsync(definition);
}
```

Generate data

A simple class is implemented in the `DataGenerator.cs` file to generate data for testing. The purpose of this class is to make it easy to generate a large number of documents with a unique ID for indexing.

To get a list of 100,000 hotels with unique IDs, run the following code:

C#

```
long numDocuments = 100000;
DataGenerator dg = new DataGenerator();
List<Hotel> hotels = dg.GetHotels(numDocuments, "large");
```

There are two sizes of hotels available for testing in this sample: *small* and *large*.

The schema of your index affects indexing speeds. After you complete this tutorial, consider converting this class to generate data that best matches your intended index schema.

Test batch sizes

To load single or multiple documents into an index, Azure AI Search supports the following APIs:

- [Documents - Index \(REST API\)](#)
- [IndexDocumentsAction class](#)
- [IndexDocumentsBatch class](#)

Indexing documents in batches significantly improves indexing performance. These batches can be up to 1,000 documents or up to about 16 MB per batch.

Determining the optimal batch size for your data is a key component of optimizing indexing speeds. The two primary factors influencing the optimal batch size are:

- The schema of your index
- The size of your data

Because the optimal batch size depends on your index and your data, the best approach is to test different batch sizes to determine what results in the fastest indexing speeds for your scenario.

The following function demonstrates a simple approach to testing batch sizes.

C#

```
public static async Task TestBatchSizesAsync(SearchClient searchClient, int min = 100, int max = 1000, int step = 100, int numTries = 3)
{
    DataGenerator dg = new DataGenerator();

    Console.WriteLine("Batch Size \t Size in MB \t MB / Doc \t Time (ms) \t MB / Second");
    for (int numDocs = min; numDocs <= max; numDocs += step)
    {
        List<TimeSpan> durations = new List<TimeSpan>();
        double sizeInMb = 0.0;
        for (int x = 0; x < numTries; x++)
        {
            List<Hotel> hotels = dg.GetHotels(numDocs, "large");

            DateTime startTime = DateTime.Now;
            await UploadDocumentsAsync(searchClient, hotels).ConfigureAwait(false);
            DateTime endTime = DateTime.Now;
            durations.Add(endTime - startTime);

            sizeInMb = EstimateObjectSize(hotels);
        }

        var avgDuration = durations.Average(TimeSpan => TimeSpan.TotalMilliseconds);
        var avgDurationInSeconds = avgDuration / 1000;
        var mbPerSecond = sizeInMb / avgDurationInSeconds;

        Console.WriteLine("{0} \t\t {1} \t\t {2} \t\t {3} \t\t {4}", numDocs,
Math.Round(sizeInMb, 3), Math.Round(sizeInMb / numDocs, 3), Math.Round(avgDuration, 3), Math.Round(mbPerSecond, 3));

        // Pausing 2 seconds to let the search service catch its breath
        Thread.Sleep(2000);
    }

    Console.WriteLine();
}
```

Because not all documents are the same size (although they are in this sample), we estimate the size of the data we're sending to the search service. You can do this by using the following function that first converts the object to JSON and then determines its size in bytes. This technique allows us to determine which batch sizes are most efficient in terms of MB/s indexing speeds.

C#

```
// Returns size of object in MB
public static double EstimateObjectSize(object data)
{
    // converting object to byte[] to determine the size of the data
    BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    byte[] Array;

    // converting data to json for more accurate sizing
    var json = JsonSerializer.Serialize(data);
    bf.Serialize(ms, json);
    Array = ms.ToArray();

    // converting from bytes to megabytes
    double sizeInMb = (double)Array.Length / 1000000;

    return sizeInMb;
}
```

The function requires a `SearchClient` plus the number of tries you'd like to test for each batch size. Because there might be variability in indexing times for each batch, try each batch three times by default to make the results more statistically significant.

C#

```
await TestBatchSizesAsync(searchClient, numTries: 3);
```

When you run the function, you should see an output in your console similar to the following example:

```

Deleting index...

Creating index...

Finding optimal batch size...

Batch Size      Size in MB      MB / Doc      Time (ms)      MB / Second
100            0.29            0.003          241.517        1.202
200            0.58            0.003          279.182        2.079
300            0.871           0.003          434.859        2.003
400            1.161           0.003          506.927        2.291
500            1.452           0.003          593.79         2.445
600            1.742           0.003          752.854        2.314
700            2.032           0.003          862.523        2.356
800            2.323           0.003          929.534        2.499
900            2.614           0.003          1082.359       2.415
1000           2.904          0.003          1255.456       2.313

Complete. Press any key to end application...

```

Identify which batch size is most efficient and use that batch size in the next step of this tutorial. You might see a plateau in MB/s across different batch sizes.

Index the data

Now that you identified the batch size you intend to use, the next step is to begin to index the data. To index data efficiently, this sample:

- Uses multiple threads/workers
- Implements an exponential backoff retry strategy

Uncomment lines 41 through 49, and then rerun the program. On this run, the sample generates and sends batches of documents, up to 100,000 if you run the code without changing the parameters.

Use multiple threads/workers

To take advantage of Azure AI Search's indexing speeds, use multiple threads to send batch indexing requests concurrently to the service.

Several of the [key considerations](#) can affect the optimal number of threads. You can modify this sample and test with different thread counts to determine the optimal thread count for your scenario. However, as long as you have several threads running concurrently, you should be able to take advantage of most of the efficiency gains.

As you ramp up the requests hitting the search service, you might encounter [HTTP status codes](#) indicating the request didn't fully succeed. During indexing, two common HTTP status codes are:

- **503 Service Unavailable:** This error means that the system is under heavy load and your request can't be processed at this time.
- **207 Multi-Status:** This error means that some documents succeeded, but at least one failed.

Implement an exponential backoff retry strategy

If a failure happens, you should retry requests using an [exponential backoff retry strategy](#).

Azure AI Search's .NET SDK automatically retries 503s and other failed requests, but you should implement your own logic to retry 207s. Open-source tools like [Polly](#) can be useful in a retry strategy.

In this sample, we implement our own exponential backoff retry strategy. We start by defining some variables, including the `maxRetryAttempts` and the initial `delay` for a failed request.

```
C#  
  
// Create batch of documents for indexing  
var batch = IndexDocumentsBatch.Upload(hotels);  
  
// Create an object to hold the result  
IndexDocumentsResult result = null;  
  
// Define parameters for exponential backoff  
int attempts = 0;  
TimeSpan delay = delay = TimeSpan.FromSeconds(2);  
int maxRetryAttempts = 5;
```

The results of the indexing operation are stored in the variable `IndexDocumentResult result`. This variable allows you to check if documents in the batch failed, as shown in the following example. If there's a partial failure, a new batch is created based on the failed documents' ID.

`RequestFailedException` exceptions should also be caught, as they indicate the request failed completely, and retried.

```
C#  
  
// Implement exponential backoff  
do  
{  
    try
```

```

{
    attempts++;
    result = await
searchClient.IndexDocumentsAsync(batch).ConfigureAwait(false);

    var failedDocuments = result.Results.Where(r => r.Succeeded != true).ToList();

    // handle partial failure
    if (failedDocuments.Count > 0)
    {
        if (attempts == maxRetryAttempts)
        {
            Console.WriteLine("[MAX RETRIES HIT] - Giving up on the batch
starting at {0}", id);
            break;
        }
        else
        {
            Console.WriteLine("[Batch starting at doc {0} had partial failure]", id);
            Console.WriteLine("[Retrying {0} failed documents] \n", failedDocuments.Count);

            // creating a batch of failed documents to retry
            var failedDocumentKeys = failedDocuments.Select(doc =>
doc.Key).ToList();
            hotels = hotels.Where(h =>
failedDocumentKeys.Contains(h.HotelId)).ToList();
            batch = IndexDocumentsBatch.Upload(hotels);

            Task.Delay(delay).Wait();
            delay = delay * 2;
            continue;
        }
    }

    return result;
}
catch (RequestFailedException ex)
{
    Console.WriteLine("[Batch starting at doc {0} failed]", id);
    Console.WriteLine("[Retrying entire batch] \n");

    if (attempts == maxRetryAttempts)
    {
        Console.WriteLine("[MAX RETRIES HIT] - Giving up on the batch starting
at {0}", id);
        break;
    }

    Task.Delay(delay).Wait();
    delay = delay * 2;
}
} while (true);

```

From here, wrap the exponential backoff code into a function so it can be easily called.

Another function is then created to manage the active threads. For simplicity, that function isn't included here but can be found in *ExponentialBackoff.cs*. You can call the function using the following command, where `hotels` is the data we want to upload, `1000` is the batch size, and `8` is the number of concurrent threads.

C#

```
await ExponentialBackoff.IndexData(indexClient, hotels, 1000, 8);
```

When you run the function, you should see an output similar to the following example:



When a batch of documents fails, an error is printed indicating the failure and that the batch is being retried.

```
[Batch starting at doc 6000 had partial failure]  
[Retrying 560 failed documents]
```

After the function finishes running, you can verify that all of the documents were added to the index.

Explore the index

After the program finishes running, you can explore the populated search index either programmatically or using the [Search explorer](#) in the Azure portal.

Programmatically

There are two main options for checking the number of documents in an index: the [Count Documents API](#) and the [Get Index Statistics API](#). Both paths require time to process, so don't be alarmed if the number of documents returned is initially lower than you expect.

Count Documents

The Count Documents operation retrieves a count of the number of documents in a search index.

C#

```
long indexDocCount = await searchClient.GetDocumentCountAsync();
```

Get Index Statistics

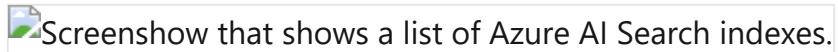
The Get Index Statistics operation returns a document count for the current index, plus storage usage. Index statistics take longer to update than document count.

C#

```
var indexStats = await indexClient.GetIndexStatisticsAsync(indexName);
```

Azure portal

In the Azure portal, from the left pane, find the **optimize-indexing** index in the **Indexes** list.



The **Document Count** and **Storage Size** are based on the [Get Index Statistics API](#) and can take several minutes to update.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure AI Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing indexes and deletes them so that you can rerun your code.

You can also use the Azure portal to delete indexes.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

Next step

To learn more about indexing large amounts data, try the following tutorial:

[Tutorial: Index large data from Apache Spark using SynapseML and Azure AI Search](#)

Last updated on 11/21/2025

Tutorial: Index large data from Apache Spark using SynapseML and Azure AI Search

In this Azure AI Search tutorial, you learn how to index and query large data loaded from a Spark cluster. You set up a Jupyter Notebook to:

- ✓ Load various forms (invoices) into a data frame in an Apache Spark session
- ✓ Analyze the forms to determine their features
- ✓ Assemble the resulting output into a tabular data structure
- ✓ Write the output to a search index hosted in Azure AI Search
- ✓ Explore and query over the content you created

This tutorial takes a dependency on [SynapseML](#)¹, an open-source library that supports massively parallel machine learning over big data. In SynapseML, search indexing and machine learning are exposed through *transformers* that perform specialized tasks. Transformers tap into a wide range of AI capabilities. In this exercise, you use the **AzureSearchWriter** APIs for analysis and AI enrichment.

Although Azure AI Search has native [AI enrichment](#), this tutorial shows you how to access AI capabilities outside of Azure AI Search. By using SynapseML instead of indexers or skills, you're not subject to data limits or other constraints associated with those objects.

💡 Tip

Watch a [short video of this demo](#)². The video expands on this tutorial with more steps and visuals.

Prerequisites

You need the `synapseml` library and several Azure resources. If possible, use the same subscription and region for your Azure resources and put everything into one resource group for simple cleanup later. The following links are for portal installs. The sample data is imported from a public site.

- [SynapseML package](#)¹
- [Azure AI Search](#) (any tier)²
- [Microsoft Foundry resource](#) (any tier) with an **API Kind** of `AI Services`³
- [Azure Databricks](#) (any tier) with Apache Spark 3.3.0 runtime⁴

¹ This link resolves to a tutorial for loading the package.

² You can use the Free tier to index the sample data, but [choose a higher tier](#) if your data volumes are large. For billable tiers, provide the [search API key](#) in the [Set up dependencies](#) step further on.

³ This tutorial uses Document Intelligence and Translator from Foundry Tools. In the instructions that follow, provide a [Foundry resource](#) key and the region. The same key works for both services. **For this tutorial, it's important that you use a Foundry resource with an API kind of AIServices.** You can check the API kind in the Azure portal on the [Overview](#) page of your Foundry resource. For more information about API kind, see [Attach a Foundry resource in Azure AI Search](#).

⁴ In this tutorial, Azure Databricks provides the Spark computing platform. We used the [portal instructions](#) to set up the cluster and workspace.

 **Note**

The preceding Azure resources support security features in the Microsoft Identity platform. For simplicity, this tutorial assumes key-based authentication, using endpoints and keys copied from the Azure portal pages of each service. If you implement this workflow in a production environment or share the solution with others, remember to replace hard-coded keys with integrated security or encrypted keys.

Create a Spark cluster and notebook

In this section, you create a cluster, install the `synapseml` library, and create a notebook to run the code.

1. In the Azure portal, find your Azure Databricks workspace and select **Launch workspace**.
2. On the left menu, select **Compute**.
3. Select **Create compute**.
4. Accept the default configuration. It takes several minutes to create the cluster.
5. Verify the cluster is operational and running. A green dot by the cluster name confirms its status.

The screenshot shows the Databricks interface. On the left, there's a sidebar with 'New', 'Workspace', and 'Recents'. The main area is titled 'Compute' and shows a search bar 'Search Demo Cluster' with a red box around it. Below the search bar are tabs for 'Configuration' (which is underlined), 'Notebooks (0)', 'Libraries', 'Event log', and 'Spark UI'.

6. After the cluster is created, install the `synapseml` library:

a. Select **Libraries** from the tabs at the top of the cluster's page.

b. Select **Install new**.

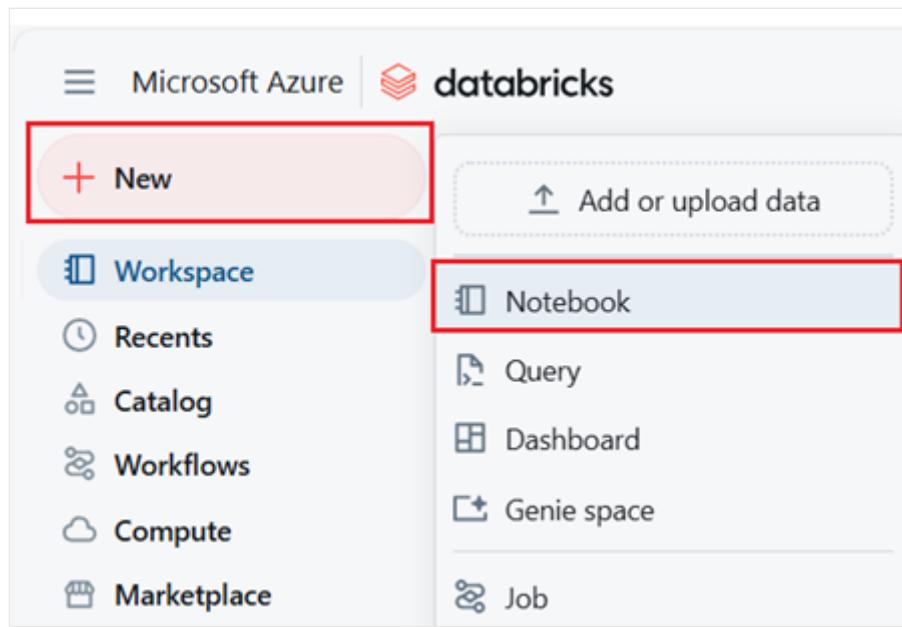


c. Select **Maven**.

d. In **Coordinates**, search for `com.microsoft.azure:synapseml_2.12:1.0.9`.

e. Select **Install**.

7. On the left menu, select **Create > Notebook**.



8. Give the notebook a name, select Python as the default language, and select the cluster that has the `synapseml` library.
9. Create seven consecutive cells. In the following sections, you paste code in these cells.



Set up dependencies

Paste the following code into the first cell of your notebook.

Replace the placeholders with endpoints and access keys for each resource. Provide a name for a new search index to be created for you. No other modifications are required, so run the code when you're ready.

This code imports multiple packages and sets up access to the Azure resources used in this tutorial.

Python

```
import os
from pyspark.sql.functions import udf, trim, split, explode, col,
monotonically_increasing_id, lit
from pyspark.sql.types import StringType
from synapse.ml.core.spark import FluentAPI

cognitive_services_key = "placeholder-azure-ai-foundry-key"
cognitive_services_region = "placeholder-azure-ai-foundry-region"

search_service = "placeholder-search-service-name"
search_key = "placeholder-search-service-admin-api-key"
search_index = "placeholder-for-new-search-index-name"
```

Load data into Spark

Paste the following code into the second cell. No modifications are required, so run the code when you're ready.

This code loads a few external files from an Azure storage account. The files are various invoices that are read into a data frame.

Python

```
def blob_to_url(blob):
    [prefix, postfix] = blob.split("@")
    container = prefix.split("/")[-1]
    split_postfix = postfix.split("/")
    account = split_postfix[0]
    filepath = "/".join(split_postfix[1:])
    return "https://{}//{}//{}//{}".format(account, container, filepath)

df2 = (spark.read.format("binaryFile")
       .load("wasbs://publicwasb@mmlspark.blob.core.windows.net/form_subset/*")
       .select("path")
       .limit(10)
       .select(udf(blob_to_url, StringType())("path").alias("url"))
       .cache())

display(df2)
```

Add document intelligence

Paste the following code into the third cell. No modifications are required, so run the code when you're ready.

This code loads the [AnalyzeInvoices transformer](#) and passes a reference to the data frame containing the invoices. It calls the prebuilt [invoice model](#) of Azure Document Intelligence in Foundry Tools to extract information from the invoices.

Python

```
from synapse.ml.services import AnalyzeInvoices

analyzed_df = (AnalyzeInvoices()
               .setSubscriptionKey(cognitive_services_key)
               .setLocation(cognitive_services_region)
               .setImageUrlCol("url")
               .setOutputCol("invoices")
               .setErrorCol("errors")
               .setConcurrency(5))
```

```

    .transform(df2)
    .cache()

display(analyzed_df)

```

The output should look similar to the following screenshot. Notice how the forms analysis is packed into a densely structured column, which is difficult to work with. The next transformation resolves this issue by parsing the column into rows and columns.

Table	Data Profile	
url	errors	invoices
https://mmlsparkdemo.blob.core.windows.net/analyzed/invoices.pdf	null	<pre> object status: "succeeded" createdDateTime: "2022-08-23T19:58:16Z" lastUpdatedDateTime: "2022-08-23T19:58:20Z" analyzeResult: version: "2.1.0" readResults: [{"page": 1, "language": null, "angle": 0, "width": 8.5, "height": 11, "unit": "inch", "lines": null}] pageResults: [{"page": 1, "keyValuePairs": null, "tables": [{"rows": 4, "columns": 8, "cells": [{"rowIndex": 0, "columnIndex": 0, "text": "Item", "boundingBox": [1.1026, 4.8201, 1.4418, 4.8201, 1.4418, 5.0852, 1.1026, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 1, "text": "Qty", "boundingBox": [1.4418, 4.8201, 1.9115, 4.8201, 1.9115, 5.0852, 1.4418, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 2, "text": "Description", "boundingBox": [1.9115, 4.8201, 4.2928, 4.8201, 4.2993, 5.0852, 1.9115, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 3, "text": "Price", "boundingBox": [4.2928, 4.8201, 4.9322, 4.8201, 4.9322, 5.0852, 4.2993, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 4, "text": "Discount", "boundingBox": [4.9322, 4.8201, 5.6433, 4.8201, 5.6498, 5.0852, 4.9322, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 5, "text": "(Pct)", "boundingBox": [5.6433, 4.8201, 6.1261, 4.8201, 6.1326, 5.0852, 5.6498, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 6, "text": "Tax", "boundingBox": [6.1261, 4.8201, 6.7263, 4.8201, 6.7263, 5.0852, 6.1326, 5.0852], "isHeader": true, "elements": null}, {"rowIndex": 0, "columnIndex": 7, "text": "LineTotal", "boundingBox": [6.7263, 4.8201, 7.4201, 4.8201, 7.4201, 5.0852, 6.7263, 5.0852], "isHeader": true, "elements": null}], "isHeader": true, "elements": null}, {"rowIndex": 1, "columnIndex": 0, "text": "1", "boundingBox": [1.1026, 4.8201, 1.4418, 4.8201, 1.4418, 5.0852, 1.1026, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 1, "text": "1", "boundingBox": [1.4418, 4.8201, 1.9115, 4.8201, 1.9115, 5.0852, 1.4418, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 2, "text": "Laptop", "boundingBox": [1.9115, 4.8201, 4.2928, 4.8201, 4.2993, 5.0852, 1.9115, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 3, "text": "100", "boundingBox": [4.2928, 4.8201, 4.9322, 4.8201, 4.9322, 5.0852, 4.2993, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 4, "text": "0", "boundingBox": [4.9322, 4.8201, 5.6433, 4.8201, 5.6498, 5.0852, 4.9322, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 5, "text": "0", "boundingBox": [5.6433, 4.8201, 6.1261, 4.8201, 6.1326, 5.0852, 5.6498, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 6, "text": "0", "boundingBox": [6.1261, 4.8201, 6.7263, 4.8201, 6.7263, 5.0852, 6.1326, 5.0852], "isHeader": false, "elements": null}, {"rowIndex": 1, "columnIndex": 7, "text": "100", "boundingBox": [6.7263, 4.8201, 7.4201, 4.8201, 7.4201, 5.0852, 6.7263, 5.0852], "isHeader": false, "elements": null}], "isHeader": false, "elements": null} </pre>

Restructure document intelligence output

Paste the following code into the fourth cell and run it. No modifications are required.

This code loads [FormOntologyLearner](#), a transformer that analyzes the output of Azure Document Intelligence transformers and infers a tabular data structure. The output of AnalyzeInvoices is dynamic and varies based on the features detected in your content. Furthermore, the transformer consolidates the output into a single column. Because the output is dynamic and consolidated, it's difficult to use in downstream transformations that require more structure.

FormOntologyLearner extends the utility of the AnalyzeInvoices transformer by looking for patterns that can be used to create a tabular data structure. Organizing the output into multiple columns and rows makes the content consumable in other transformers, like AzureSearchWriter.

Python

```

from synapse.ml.cognitive import FormOntologyLearner

itemized_df = (FormOntologyLearner()
    .setInputCol("invoices")
    .setOutputCol("extracted"))

```

```

    .fit(analyzed_df)
    .transform(analyzed_df)
    .select("url", "extracted.*").select("*", explode(col("Items")).alias("Item"))
    .drop("Items").select("Item.*", "*").drop("Item"))

display(itemized_df)

```

Notice how this transformation recasts the nested fields into a table, which enables the next two transformations. This screenshot is trimmed for brevity. If you're following along in your own notebook, you have 19 columns and 26 rows.

Table Data Profile							
	ProductCode	Tax	Quantity	UnitPrice	Description	Amount	url
1	61	3.78	1	54	One sat on shoe gnome	57.78	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
2	23	17.22	2	123	White Door	263.22	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
3	42	5.67	1	90	Metal Shelving	86.67	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
4	35	4.2	2	30	Wood Pack	64.2	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
5	45	30.45	3	145	Wooden Saw	465.45	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
6	23	25.83	3	123	White Door	394.83	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
7							

Showing all 26 rows.

Add translations

Paste the following code into the fifth cell. No modifications are required, so run the code when you're ready.

This code loads [Translate](#), a transformer that calls Azure Translator in Foundry Tools. The original text, which is in English in the "Description" column, is machine-translated into various languages. All of the output is consolidated into the "output.translations" array.

Python

```

from synapse.ml.cognitive import Translate

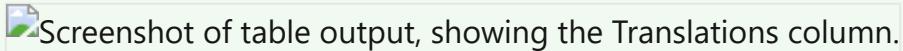
translated_df = (Translate()
    .setSubscriptionKey(cognitive_services_key)
    .setLocation(cognitive_services_region)
    .setTextCol("Description")
    .setErrorCol("TranslationError")
    .setOutputCol("output")
    .setToLanguage(["zh-Hans", "fr", "ru", "cy"])
    .setConcurrency(5)
    .transform(itemized_df)
    .withColumn("Translations", col("output.translations")[0])
    .drop("output", "TranslationError")
    .cache())

display(translated_df)

```

💡 Tip

To check for translated strings, scroll to the end of the rows.



Add a search index with AzureSearchWriter

Paste the following code in the sixth cell and run it. No modifications are required.

This code loads [AzureSearchWriter](#). It consumes a tabular dataset and infers a search index schema that defines one field for each column. Because the translations structure is an array, it's articulated in the index as a complex collection with subfields for each language translation. The generated index has a document key and uses the default values for fields created using the [Create Index REST API](#).

Python

```
from synapse.ml.cognitive import *

(translated_df.withColumn("DocID", monotonically_increasing_id().cast("string"))
 .withColumn("SearchAction", lit("upload"))
 .writeToAzureSearch(
     subscriptionKey=search_key,
     actionCol="SearchAction",
     serviceName=search_service,
     indexName=search_index,
     keyCol="DocID",
 ))
```

To explore the index definition created by AzureSearchWriter, check the search service pages in the Azure portal.

⚠ Note

If you can't use the default search index, you can provide an external custom definition in JSON, passing its URL as a string in the "indexJson" property. Generate the default index first so that you know which fields to specify, and then follow with customized properties if you need specific analyzers, for example.

Query the index

Paste the following code into the seventh cell and run it. No modifications are required, except that you might want to vary the syntax or try more examples to further explore your content:

- [Query syntax](#)
- [Query examples](#)

There's no transformer or module that issues queries. This cell is a simple call to the [Search Documents REST API](#).

This particular example is searching for the word "door" (`"search": "door"`). It also returns a "count" of the number of matching documents and selects just the contents of the "Description" and "Translations" fields for the results. If you want to see the full list of fields, remove the "select" parameter.

Python

```
import requests

url = "https://{}.search.windows.net/indexes/{}/docs/search?api-version=2025-09-01".format(search_service, search_index)
requests.post(url, json={"search": "door", "count": "true", "select": "Description, Translations"}, headers={"api-key": search_key}).json()
```

The following screenshot shows the cell output for sample script.



Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the [All resources](#) or [Resource groups](#) link in the left-navigation pane.

Next steps

In this tutorial, you learned about the [AzureSearchWriter](#) transformer in SynapseML, which is a new way of creating and loading search indexes in Azure AI Search. The transformer takes structured JSON as an input. FormOntologyLearner can provide the necessary structure for output produced by the Azure Document Intelligence transformers in SynapseML.

As a next step, review the other SynapseML tutorials that produce transformed content you might want to explore through Azure AI Search:

Tutorial: Text Analytics with Foundry Tools

Last updated on 11/18/2025

Tutorial: Index Azure SQL data using the .NET SDK

Article • 04/09/2025

Learn how to configure an [indexer](#) to extract searchable data from Azure SQL Database and send it to a search index in Azure AI Search.

In this tutorial, you use C# and the [Azure SDK for .NET](#) to:

- ✓ Create a data source that connects to Azure SQL Database
- ✓ Create an indexer
- ✓ Run an indexer to load data into an index
- ✓ Query an index as a verification step

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure SQL Database](#) using SQL Server authentication.
- [Azure AI Search](#). [Create a service](#) or [find an existing service](#) in your current subscription.
- [Visual Studio](#).

ⓘ Note

You can use a free search service for this tutorial. The Free tier limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before you start, make sure you have room on your service to accept the new resources.

Download files

Source code for this tutorial is in the [DotNetHowToIndexer](#) folder in the [Azure-Samples/search-dotnet-getting-started](#) GitHub repository.

Create services

This tutorial uses Azure AI Search for indexing and queries and Azure SQL Database as an external data source. If possible, create both services in the same region and resource group for proximity and manageability. In practice, Azure SQL Database can be in any region.

Start with Azure SQL Database

This tutorial provides the *hotels.sql* file in the sample download to populate the database. Azure AI Search consumes flattened rowsets, such as one generated from a view or query. The SQL file in the sample solution creates and populates a single table.

If you have an existing Azure SQL Database resource, you can add the hotels table to it starting at the **Open query** step.

1. Create an Azure SQL database.

Server configuration for the database is important:

- Choose the SQL Server authentication option that prompts you to specify a username and password. You need this for the ADO.NET connection string used by the indexer.
- Choose a public connection, which makes this tutorial easier to complete. Public isn't recommended for production, and we recommend [deleting this resource](#) at the end of the tutorial.

The screenshot shows the 'Create SQL Database' wizard in the Azure portal. The 'Networking' tab is selected. In the 'Connectivity method' section, the 'Public endpoint' option is selected and highlighted with a red box. In the 'Firewall rules' section, the 'Allow Azure services and resources to access this server' option is set to 'Yes' and highlighted with a red box. The 'Add current client IP address' option is also set to 'Yes'.

2. In the Azure portal, go to the new resource.

3. Add a firewall rule that allows access from your client. You can run `ipconfig` from a command prompt to get your IP address.

4. Use the Query editor to load the sample data. On the navigation pane, select **Query editor (preview)** and enter the username and password of the server admin.

If you get an access denied error, copy the client IP address from the error message, open the network security page for the server, and add an inbound rule that allows access from your client.

5. In Query editor, select **Open query** and navigate to the location of *hotels.sql* file on your local computer.

6. Select the file and select **Open**. The script should look similar to the following screenshot:

```
Query 1 × Query 2 ×

Run Cancel query Save query Export data as .json ...

1 ALTER DATABASE CURRENT
2 SET CHANGE_TRACKING = ON
3 (CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON)
4
5 CREATE TABLE Hotels (
6     [HotelId] nvarchar(450) NOT NULL PRIMARY KEY,
7     [BaseRate] float NULL,
8     [Category] nvarchar(max) NULL,
9     [Description] nvarchar(max) NULL,
10    [Description_fr] nvarchar(max) NULL,
11    [HotelName] nvarchar(max) NULL,
12    [Tags] nvarchar(max) NULL,
13    [IsDeleted] bit NOT NULL,
14    [LastRenovationDate] DateTime NULL,
)

Results Messages

Query succeeded: Affected rows: 0Affected rows: 3
```

7. Select **Run** to execute the query. In the **Results** pane, you should see a query succeeded message for three rows.

8. To return a rowset from this table, you can execute the following query as a verification step:

```
SQL

SELECT * FROM Hotels
```

9. Copy the ADO.NET connection string for the database. Under **Settings > Connection Strings**, copy the ADO.NET connection string, which should be similar to the following example:

SQL

```
Server=tcp:<YOUR-DATABASE-NAME>.database.windows.net,1433;Initial Catalog=hotels-db;Persist Security Info=False;User ID=<YOUR-USER-NAME>;Password=<YOUR-PASSWORD>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
```

You'll need this connection string to set up your environment in the next step.

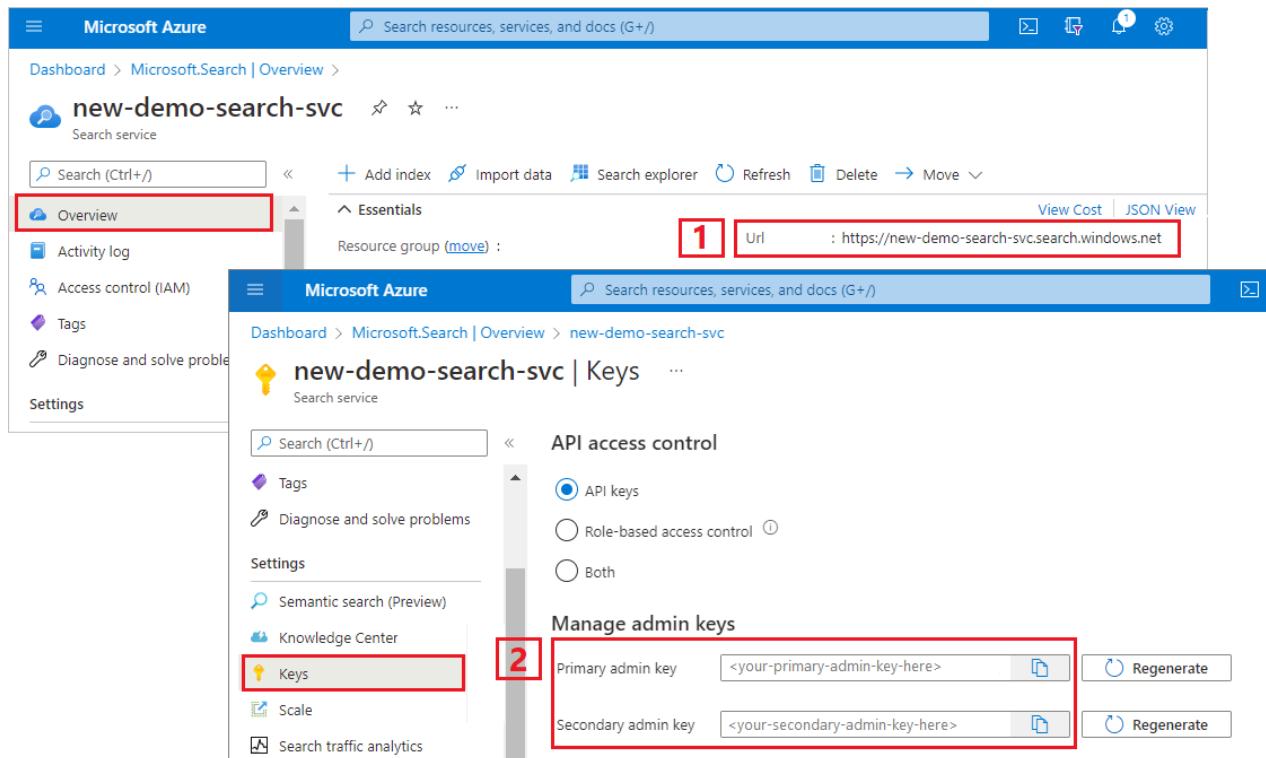
Azure AI Search

The next component is Azure AI Search, which you can [create in the Azure portal](#). You can use the Free tier to complete this tutorial.

Get an admin key and URL for Azure AI Search

API calls require the service URL and an access key. A search service is created with both, so if you added Azure AI Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the [Azure portal](#). On your service **Overview** page, copy the endpoint URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. On **Settings > Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.



Set up your environment

1. Start Visual Studio and open `DotNetHowToIndexers.sln`.
2. In Solution Explorer, open `appsettings.json` to provide connection information.
3. For `SearchServiceEndPoint`, if the full URL on your service **Overview** page is `https://my-demo-service.search.windows.net`, provide the entire URL.
4. For `AzureSqlConnectionString`, the string format is similar to `"Server=tcp:<your-database-name>.database.windows.net,1433;Initial Catalog=hotels-db;Persist Security Info=False;User ID=<your-user-name>;Password=<your-password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"`.

```
JSON

{
  "SearchServiceEndPoint": "<placeholder-search-full-url>",
  "SearchServiceAdminApiKey": "<placeholder-admin-key-for-search-service>",
  "AzureSqlConnectionString": "<placeholder-ADO.NET-connection-string>",
}
```

5. Replace the user password in the SQL connection string with a valid password. While the database and usernames will copy over, you must enter the password manually.

Create the pipeline

Indexers require a data source object and an index. The relevant code is in two files:

- *hotel.cs* contains a schema that defines the index
- *Program.cs* contains functions for creating and managing structures in your service

In hotel.cs

The index schema defines the fields collection, including attributes specifying allowed operations, such as whether a field is full-text searchable, filterable, or sortable, as shown in the following field definition for `HotelName`. A [SearchableField](#) is, by definition, full-text searchable. Other attributes are explicitly assigned.

```
C#  
.  
.  
[SearchableField(IsFilterable = true, IsSortable = true)]  
[JsonPropertyName("hotelName")]  
public string HotelName { get; set; }  
.  
.
```

A schema can also include other elements, such as scoring profiles for boosting a search score and custom analyzers. However, for this tutorial, the schema is sparsely defined, consisting only of fields found in the sample datasets.

In Program.cs

The main program includes logic for creating [an indexer client](#), an index, a data source, and an indexer. The code checks for and deletes existing resources of the same name, assuming that you might run this program multiple times.

The data source object is configured with settings that are specific to Azure SQL Database resources, including [partial or incremental indexing](#) for using the built-in [change detection features](#) of Azure SQL. The source demo hotels database in Azure SQL has a "soft delete" column named `IsDeleted`. When this column is set to true in the database, the indexer removes the corresponding document from the Azure AI Search index.

```
C#  
  
Console.WriteLine("Creating data source...");  
  
var dataSource =  
    new SearchIndexerDataSourceConnection(  
        . . . );
```

```

    "hotels-sql-ds",
    SearchIndexerDataSourceType.AzureSql,
    configuration[ "AzureSQLConnectionString" ],
    new SearchIndexerDataContainer( "hotels" ));

indexerClient.CreateOrUpdateDataSourceConnection(dataSource);

```

An indexer object is platform agnostic, where configuration, scheduling, and invocation are the same regardless of the source. This example indexer includes a schedule and a reset option that clears the indexer history. It also calls a method to create and run the indexer immediately. To create or update an indexer, use [CreateOrUpdateIndexerAsync](#).

C#

```

Console.WriteLine("Creating Azure SQL indexer...");

var schedule = new IndexingSchedule(TimeSpan.FromDays(1))
{
    StartTime = DateTimeOffset.Now
};

var parameters = new IndexingParameters()
{
    BatchSize = 100,
    MaxFailedItems = 0,
    MaxFailedItemsPerBatch = 0
};

// Indexer declarations require a data source and search index.
// Common optional properties include a schedule, parameters, and field mappings
// The field mappings below are redundant due to how the Hotel class is defined,
// but
// we included them anyway to show the syntax
var indexer = new SearchIndexer("hotels-sql-idxr", dataSource.Name,
searchIndex.Name)
{
    Description = "Data indexer",
    Schedule = schedule,
    Parameters = parameters,
    FieldMappings =
    {
        new FieldMapping("_id") {TargetFieldName = "HotelId"},
        new FieldMapping("Amenities") {TargetFieldName = "Tags"}
    }
};

await indexerClient.CreateOrUpdateIndexerAsync(indexer);

```

Indexer runs are usually scheduled, but during development, you might want to run the indexer immediately using [RunIndexerAsync](#).

C#

```
Console.WriteLine("Running Azure SQL indexer...");

try
{
    await indexerClient.RunIndexerAsync(indexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 429)
{
    Console.WriteLine("Failed to run indexer: {0}", ex.Message);
}
```

Build the solution

Select F5 to build and run the solution. The program executes in debug mode. A console window reports the status of each operation.

```
Creating index...
Creating data source...
Creating Azure SQL indexer...
Running Azure SQL indexer...
Press any key to continue...
```

Your code runs locally in Visual Studio, connecting to your search service on Azure, which in turn connects to Azure SQL Database and retrieves the dataset. With this many operations, there are several potential points of failure. If you get an error, check the following conditions first:

- Search service connection information that you provide is the full URL. If you only entered the service name, operations stop at index creation, with a failure to connect error.
- Database connection information in *appsettings.json*. It should be the ADO.NET connection string obtained from the Azure portal, modified to include a username and password that are valid for your database. The user account must have permission to retrieve data. Your local client IP address must be allowed inbound access through the firewall.
- Resource limits. Recall that the Free tier has limits of three indexes, indexers, and data sources. A service at the maximum limit can't create new objects.

Search

Use the Azure portal to verify object creation, and then use **Search explorer** to query the index.

1. Sign in to the [Azure portal](#) and go to your search service. From the left pane, open each page to verify the objects are created. **Indexes**, **Indexers**, and **Data Sources** should have **hotels-sql-idx**, **hotels-sql-indexer**, and **hotels-sql-ds**, respectively.
2. On the **Indexes** tab, select the **hotels-sql-idx** index. On the hotels page, **Search explorer** is the first tab.
3. Select **Search** to issue an empty query.

The three entries in your index are returned as JSON documents. Search explorer returns documents in JSON so that you can view the entire structure.

The screenshot shows the Azure Search Explorer interface for the 'hotels-sql-idx' index. The top navigation bar includes 'Dashboard > Indexes > hotels-sql-idx'. Below the navigation is a toolbar with 'Save', 'Discard', 'Refresh', 'Create Demo App', 'Edit JSON', and 'Delete' buttons. The main area displays document statistics: 'Documents' (3) and 'Storage' (19.77 KB). A navigation bar below the stats includes tabs for 'Search explorer' (which is selected and highlighted with a red box), 'Fields', 'CORS', 'Scoring profiles', 'Semantic configurations', and 'Vector profiles'. To the right of the tabs are 'Query options' and 'View' dropdowns. A search bar with a magnifying glass icon and a 'Search' button are located at the bottom of the main area. The 'Results' section contains a table with 35 rows of JSON document data. The first few rows are:

Line Number	Document Content
18	{
19	"@search.score": 1,
20	"hotelID": "2",
21	"baseRate": 79.99,
22	"description": "Cheapest hotel in town",
23	"description_fr": "Hôtel le moins cher en ville",
24	"hotelName": "Roach Motel",
25	"category": "Budget",
26	"tags": [
27	"motel",
28	"budget"
29],
30	"parkingIncluded": true,
31	"smokingAllowed": true,
32	"lastRenovationDate": "1982-04-28T00:00:00Z",
33	"rating": 1
34	},
35	{

4. Switch to **JSON view** so that you can enter query parameters.

The screenshot shows the Azure Search Explorer interface with the 'JSON' tab selected. The JSON input field contains the following search query:

```
{
  "search": "river",
  "count": true
}
```

This query invokes full text search on the term `river`. The result includes a count of the matching documents. Returning the count of matching documents is helpful in testing scenarios where you have a large index with thousands or millions of documents. In this case, only one document matches the query.

5. Enter parameters that limit search results to fields of interest.

JSON

```
{  
  "search": "river",  
  "select": "hotelId, hotelName, baseRate, description",  
  "count": true  
}
```

The query response is reduced to selected fields, resulting in more concise output.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure AI Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing objects and deletes them so that you can rerun your code.

You can also use the Azure portal to delete indexes, indexers, and data sources.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with the basics of SQL Database indexing, take a closer look at indexer configuration:

Configure a SQL Database indexer

Tutorial: Index nested JSON blobs from Azure Storage using REST

Azure AI Search can index JSON documents and arrays in Azure Blob Storage using an [indexer](#) that knows how to read semi-structured data. Semi-structured data contains tags or markings that separate content within the data. It splits the difference between unstructured data, which must be fully indexed, and formally structured data that adheres to a data model, such as a relational database schema that can be indexed on a per-field basis.

This tutorial shows you how to index nested JSON arrays, using a REST client and the [Search REST APIs](#) to:

- ✓ Set up sample data and configure an `azureblob` data source
- ✓ Create an Azure AI Search index to contain searchable content
- ✓ Create and run an indexer to read the container and extract searchable content
- ✓ Search the index you just created

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure Storage](#).
- [Azure AI Search](#). [Create a service](#) or [find an existing service](#) in your current subscription.
- [Visual Studio Code](#) with a [REST client](#).

Note

You can use a free search service for this tutorial. The Free tier limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before you start, make sure you have room on your service to accept the new resources.

Download files

Download a zip file of the sample data repository and extract the contents. [Learn how](#).

- [ny-philharmonic-free](#)

The sample data is a single JSON file that contains a JSON array and 1,521 nested JSON elements. The data originates from the [NY Philharmonic Performance History](#) on Kaggle. We

chose one JSON file to stay under the storage limits of the Free tier.

Here's the first nested JSON in the file. The remainder of the file includes 1,520 other instances of concert performances.

JSON

```
{  
  "id": "7358870b-65c8-43d5-ab56-514bde52db88-0.1",  
  "programID": "11640",  
  "orchestra": "New York Philharmonic",  
  "season": "2011-12",  
  "concerts": [  
    {  
      "eventType": "Non-Subscription",  
      "Location": "Manhattan, NY",  
      "Venue": "Avery Fisher Hall",  
      "Date": "2011-09-07T04:00:00Z",  
      "Time": "7:30PM"  
    },  
    {  
      "eventType": "Non-Subscription",  
      "Location": "Manhattan, NY",  
      "Venue": "Avery Fisher Hall",  
      "Date": "2011-09-08T04:00:00Z",  
      "Time": "7:30PM"  
    }  
,  
  ],  
  "works": [  
    {  
      "ID": "5733*",  
      "composerName": "Bernstein, Leonard",  
      "workTitle": "WEST SIDE STORY (WITH FILM)",  
      "conductorName": "Newman, David",  
      "soloists": []  
    },  
    {  
      "ID": "0*",  
      "interval": "Intermission",  
      "soloists": []  
    }  
  ]  
}
```

Upload sample data to Azure Storage

1. In Azure Storage, create a new container named **ny-philharmonic-free**.
2. [Upload the sample data files](#).

3. Get a storage connection string so that you can formulate a connection in Azure AI Search.

a. On the left, select **Access keys**.

b. Copy the connection string for either key one or key two. The connection string is similar to the following example:

HTTP

```
DefaultEndpointsProtocol=https;AccountName=<your account name>;AccountKey=<your account key>;EndpointSuffix=core.windows.net
```

Copy a search service URL and API key

For this tutorial, connections to Azure AI Search require an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Managed identities](#).

1. Sign in to the [Azure portal](#) and select your search service.

2. From the left pane, select **Overview** and copy the endpoint. It should be in this format:

```
https://my-service.search.windows.net
```

3. From the left pane, select **Settings > Keys** and copy an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests to add, modify, or delete objects.

The screenshot displays two separate windows of the Microsoft Azure portal. The top window is titled 'new-demo-search-svc' and shows the 'Overview' tab selected. A red box highlights the 'Overview' tab, and another red box highlights the URL 'https://new-demo-search-svc.search.windows.net'. The bottom window is also titled 'new-demo-search-svc' and shows the 'Keys' tab selected. A red box highlights the 'Keys' tab, and another red box highlights the 'Primary admin key' and 'Secondary admin key' fields, which both contain placeholder text '<your-primary-admin-key-here>' and '<your-secondary-admin-key-here>'. Both of these fields have 'Regenerate' buttons to their right.

Set up your REST file

1. Start Visual Studio Code and create a new file.
2. Provide values for variables used in the request.

```
HTTP

@baseUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE
@apiKey = PUT-YOUR-ADMIN-API-KEY-HERE
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE
@blobContainer = PUT-YOUR-CONTAINER-NAME-HERE
```

3. Save the file using a `.rest` or `.http` file extension.

For help with the REST client, see [Quickstart: Full-text search using REST](#).

Create a data source

[Create Data Source \(REST\)](#) creates a data source connection that specifies what data to index.

```
HTTP

### Create a data source
POST {{baseUrl}}/datasources?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}
```

```
{
    "name" : "ny-philharmonic-ds",
    "description": null,
    "type": "azureblob",
    "subtype": null,
    "credentials": {
        "connectionString": "{{storageConnection}}"
    },
    "container": {
        "name": "{{blobContainer}}",
        "query": null
    },
    "dataChangeDetectionPolicy": null,
    "dataDeletionDetectionPolicy": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
ETag: "0x8DC43A5FDB8448F"
Location: https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net:443/datasources('ny-philharmonic-ds')?api-version=2025-09-01
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 7ca53f73-1054-4959-bc1f-616148a9c74a
elapsed-time: 111
Date: Wed, 13 Mar 2024 21:38:58 GMT
Connection: close

{
    "@odata.context": "https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/$metadata#datasources/$entity",
    "@odata.etag": "\"0x8DC43A5FDB8448F\"",
    "name": "ny-philharmonic-ds",
    "description": null,
    "type": "azureblob",
    "subtype": null,
    "credentials": {
        "connectionString": null
    },
    "container": {
        "name": "ny-philharmonic-free",
        "query": null
    },
    "dataChangeDetectionPolicy": null,
    "dataDeletionDetectionPolicy": null,
```

```
        "encryptionKey": null  
    }
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the parameters and their attributes.

For nested JSON, the index fields must be identical to the source fields. Currently, Azure AI Search doesn't support field mappings to nested JSON, so field names and data types must match completely. The following index aligns to the JSON elements in the raw content.

HTTP

```
### Create an index  
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1  
Content-Type: application/json  
api-key: {{apiKey}}  
  
{  
    "name": "ny-philharmonic-index",  
    "fields": [  
        {"name": "programID", "type": "Edm.String", "key": true, "searchable": true,  
        "retrievable": true, "filterable": true, "facetable": true, "sortable": true},  
        {"name": "orchestra", "type": "Edm.String", "searchable": true,  
        "retrievable": true, "filterable": true, "facetable": true, "sortable": true},  
        {"name": "season", "type": "Edm.String", "searchable": true, "retrievable":  
        true, "filterable": true, "facetable": true, "sortable": true},  
        {"name": "concerts", "type": "Collection(Edm.ComplexType)",  
        "fields": [  
            { "name": "eventType", "type": "Edm.String", "searchable": true,  
            "retrievable": true, "filterable": false, "sortable": false, "facetable": false},  
            { "name": "Location", "type": "Edm.String", "searchable": true,  
            "retrievable": true, "filterable": true, "sortable": false, "facetable": true },  
            { "name": "Venue", "type": "Edm.String", "searchable": true,  
            "retrievable": true, "filterable": true, "sortable": false, "facetable": true },  
            { "name": "Date", "type": "Edm.String", "searchable": false,  
            "retrievable": true, "filterable": true, "sortable": false, "facetable": true },  
            { "name": "Time", "type": "Edm.String", "searchable": false,  
            "retrievable": true, "filterable": true, "sortable": false, "facetable": true }  
        ]  
    ],  
    { "name": "works", "type": "Collection(Edm.ComplexType)",  
    "fields": [  
        { "name": "ID", "type": "Edm.String", "searchable": true, "retrievable":  
        true, "filterable": false, "sortable": false, "facetable": false},  
        { "name": "composerName", "type": "Edm.String", "searchable": true,  
        "retrievable": true, "filterable": true, "sortable": false, "facetable": true },  
        { "name": "workTitle", "type": "Edm.String", "searchable": true,  
        "retrievable": true, "filterable": true, "sortable": false, "facetable": true },  
        { "name": "conductorName", "type": "Edm.String", "searchable": true,
```

```

    "retrievable": true, "filterable": true, "sortable": false, "facetable": true },
        { "name": "soloists", "type": "Collection(Edm.String)", "searchable": true,
        "retrievable": true, "filterable": true, "sortable": false, "facetable": true }
    }
]
}
}

```

Key points:

- You can't use [field mappings](#) to reconcile differences in field names or data types. This index schema is designed to mirror the raw content.
- Nested JSON is modeled as `Collection(Edm.ComplexType)`. In the raw content, there are multiple concerts for each season, and multiple works for each concert. To accommodate this structure, use collections for complex types.
- In the raw content, `Date` and `Time` are strings, so the corresponding data types in the index are also strings.

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads and indexes data, and optionally provides a schedule to automate the data refresh.

The indexer configuration includes the `jsonArray` parsing mode and a `documentRoot`.

HTTP

```

### Create and run an indexer
POST {{baseUrl}}/indexers?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
    "name" : "ny-philharmonic-indexer",
    "dataSourceName" : "ny-philharmonic-ds",
    "targetIndexName" : "ny-philharmonic-index",
    "parameters" : {
        "configuration" : {
            "parsingMode" : "jsonArray", "documentRoot": "/programs"
        },
        "fieldMappings" : [
        ]
    }
}

```

Key points:

- The raw content file contains a JSON array ("programs") with 1,526 nested JSON structures. Set `parsingMode` to `jsonArray` to tell the indexer that each blob contains a JSON array. Because the nested JSON starts one level down, set `documentRoot` to `/programs`.
- The indexer runs for several minutes. Wait for indexer execution to complete before running any queries.

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```
### Query the index
POST {{baseUrl}}/indexes/ny-philharmonic-index/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

{
  "search": "*",
  "count": true
}
```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: a95c4021-f7b4-450b-ba55-596e59ecb6ec
elapsed-time: 106
Date: Wed, 13 Mar 2024 22:09:59 GMT
Connection: close
```

```
{  
    "@odata.context": "https://<YOUR-SEARCH-SERVICE-  
NAME>.search.windows.net/indexes('ny-philharmonic-index')/$metadata#docs(*)",  
    "@odata.count": 1521,  
    "@search.nextPageParameters": {  
        "search": "*",  
        "count": true,  
        "skip": 50  
    },  
    "value": [  
    ],  
    "@odata.nextLink": "https://<YOUR-SEARCH-SERVICE-  
NAME>.search.windows.net/indexes/ny-philharmonic-index/docs/search?api-version=2025-  
09-01"  
}
```

Add a `search` parameter to search on a string, a `select` parameter to limit the results to fewer fields, and a `filter` to further narrow the search.

HTTP

```
### Query the index  
POST {{baseUrl}}/indexes/ny-philharmonic-index/docs/search?api-version=2025-09-01  
HTTP/1.1  
Content-Type: application/json  
api-key: {{apiKey}}  
  
{  
    "search": "puccini",  
    "count": true,  
    "select": "season, concerts/Date, works/composerName, works/workTitle",  
    "filter": "season gt '2015-16'"  
}
```

Two documents are returned in the response.

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case -sensitive. For more information and examples, see [Create a query](#).

ⓘ Note

The `$filter` parameter only works on fields that were marked filterable during index creation.

Reset and rerun

Indexers can be reset to clear execution history, which allows a full rerun. The following POST requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{baseUrl}}/indexers/ny-philharmonic-indexer/reset?api-version=2025-09-01
HTTP/1.1
    api-key: {{apiKey}}
```

HTTP

```
### Run the indexer
POST {{baseUrl}}/indexers/ny-philharmonic-indexer/run?api-version=2025-09-01
HTTP/1.1
    api-key: {{apiKey}}
```

HTTP

```
### Check indexer status
GET {{baseUrl}}/indexers/ny-philharmonic-indexer/status?api-version=2025-09-01
HTTP/1.1
    api-key: {{apiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

Next steps

Now that you're familiar with the basics of Azure Blob indexing, take a closer look at indexer configuration for JSON blobs in Azure Storage:

[Configure JSON blob indexing](#)

Tutorial: Index nested Markdown blobs from Azure Storage using REST

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Azure AI Search can index Markdown documents and arrays in Azure Blob Storage using an [indexer](#) that knows how to read Markdown data.

This tutorial shows you how to index Markdown files indexed using the `oneToMany` Markdown parsing mode and the [Search Service REST APIs](#).

In this tutorial, you:

- ✓ Set up sample data and configure an `azureblob` data source
- ✓ Create an Azure AI Search index to contain searchable content
- ✓ Create and run an indexer to read the container and extract searchable content
- ✓ Search the index you just created

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure Storage account](#).
- An [Azure AI Search service](#).
- [Visual Studio Code](#) with the [REST Client Extension](#).

! Note

You can use a free search service for this tutorial. The Free tier limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before you start, make sure your service has room to accept the new resources.

Prepare sample data

Create a Markdown file

Copy and paste the following Markdown into a file named `sample_markdown.md`. The sample data is a single Markdown file containing various Markdown elements. We chose one Markdown file to stay under the storage limits of the Free tier.

Markdown

Project Documentation

Introduction

This document provides a complete overview of the **Markdown Features** used within this project. The following sections demonstrate the richness of Markdown formatting, with examples of lists, tables, links, images, blockquotes, inline styles, and more.

Table of Contents

1. [Headers](#headers)
2. [Introduction](#introduction)
3. [Basic Text Formatting](#basic-text-formatting)
4. [Lists](#lists)
5. [Blockquotes](#blockquotes)
6. [Images](#images)
7. [Links](#links)
8. [Tables](#tables)
9. [Code Blocks and Inline Code](#code-blocks-and-inline-code)
10. [Horizontal Rules](#horizontal-rules)
11. [Inline Elements](#inline-elements)
12. [Escaping Characters](#escaping-characters)
13. [HTML Elements](#html-elements)
14. [Emojis](#emojis)
15. [Footnotes](#footnotes)
16. [Task Lists](#task-lists)
17. [Conclusion](#conclusion)

Headers

Markdown supports six levels of headers. Use `#` to create headers:

"# Project Documentation" at the top of the document is an example of an h1 header.

"## Headers" above is an example of an h2 header.

h3 example

h4 example

h5 example

h6 example

This is an example of content underneath a header.

Basic Text Formatting

You can apply various styles to your text:

- **Bold**: Use double asterisks or underscores: `**bold**` or `__bold__`.
- *Italic*: Use single asterisks or underscores: `*italic*` or `_italic_`.

- ~~Strikethrough~~: Use double tildes: `~~strikethrough~~`.

Lists

Ordered List

1. First item
2. Second item
3. Third item

Unordered List

- Item A
- Item B
- Item C

Nested List

1. Parent item
 - Child item
 - Child item

Blockquotes

> This is a blockquote.
> Blockquotes are great for emphasizing important information.
>> Nested blockquotes are also possible!

Images

![Markdown Logo](<https://markdown-here.com/img/icon256.png>)

Links

[Visit Markdown Guide](<https://www.markdownguide.org>)

Tables

Syntax	Description	Example
Header	Title	Header Cell
Paragraph	Text block	Row Content

Code Blocks and Inline Code

Inline Code

Use backticks to create `inline code`.

Code Block

```
```javascript
// JavaScript example
function greet(name) {
 console.log(`Hello, ${name}!`);
}
greet('World');
```
```

Horizontal Rules

Use three or more dashes or underscores to create a horizontal rule.

Inline Elements

Sometimes, it's useful to include `inline code` to highlight code-like content.

You can also emphasize text like *this* or make it **bold**.

Escaping Characters

To render special Markdown characters, use backslashes:

- *Asterisks*
- \#Hashes\#
- \[Brackets\]

HTML Elements

You can mix HTML tags with Markdown:

```
<table>
  <tr>
    <th>HTML Table</th>
    <th>With Markdown</th>
  </tr>
  <tr>
    <td>Row 1</td>
    <td>Data 1</td>
  </tr>
</table>
```

Emojis

Markdown supports some basic emojis:

- :smile: 😊
- :rocket: 🚀
- :checkered_flag: 🇩🇪

Footnotes

This is an example of a footnote^[^1]. Footnotes allow you to add notes without cluttering the main text.

[^1]: This is the content of the footnote.

Task Lists

- [x] Complete the introduction
- [] Add more examples
- [] Review the document

Conclusion

Markdown is a lightweight yet powerful tool for writing documentation. It supports a variety of formatting options while maintaining simplicity and readability.

Thank you for reviewing this example!

Upload the file and get a connection string

Follow these instructions to upload the `sample_markdown.md` file to a container in your Azure Storage account. You must also get the storage account connection string. Make a note of the connection string and the container name for later use.

Copy a search service URL and API key

For this tutorial, connections to Azure AI Search require an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Managed identities](#).

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview**.
3. Make a note of the URL, which should look like `https://my-service.search.windows.net`.
4. From the left pane, select **Settings > Keys**.
5. Make a note of an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests for adding, modifying, and deleting objects.



Screenshot of the URL and API keys in the Azure portal.

Set up your REST file

1. Create a file in Visual Studio Code.
2. Provide values for variables used in the request.

HTTP

```
@baseUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE  
apiKey = PUT-YOUR-ADMIN-API-KEY-HERE  
@storageConnectionString = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE  
@blobContainer = PUT-YOUR-CONTAINER-NAME-HERE
```

3. Save the file using a `.rest` or `.http` file extension.

For help with the REST client, see [Quickstart: Full-text search using REST](#).

Create a data source

[Create Data Source \(REST\)](#) creates a data source connection that specifies what data to index.

HTTP

```
### Create a data source
POST {{baseUrl}}/datasources?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
    "name" : "sample-markdown-ds",
    "description": null,
    "type": "azureblob",
    "subtype": null,
    "credentials": {
        "connectionString": "{{storageConnectionString}}"
    },
    "container": {
        "name": "{{blobContainer}}",
        "query": null
    },
    "dataChangeDetectionPolicy": null,
    "dataDeletionDetectionPolicy": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
ETag: "0x8DCF52E926A3C76"
Location: https://<YOUR-SEARCH-SERVICE-
NAME>.search.windows.net:443/datasources('sample-markdown-ds')?api-version=2025-09-
01
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 0714c187-217e-4d35-928a-5069251e5cba
elapsed-time: 204
Date: Fri, 25 Oct 2024 19:52:35 GMT
Connection: close

{
    "@odata.context": "https://<YOUR-SEARCH-SERVICE-
NAME>.search.windows.net/$metadata#datasources/$entity",
    "@odata.etag": "\"0x8DCF52E926A3C76\"",
    "name": "sample-markdown-ds",
    "description": null,
    "type": "azureblob",
    "subtype": null,
    "credentials": {
```

```
    "connectionString": null
},
"container": {
    "name": "markdown-container",
    "query": null
},
"dataChangeDetectionPolicy": null,
"dataDeletionDetectionPolicy": null,
"encryptionKey": null,
"identity": null
}
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the fields and their attributes.

In one-to-many parsing, the search document defines the 'many' side of the relationship. The fields you specify in the index determine the structure of the search document.

You only need fields for the Markdown elements that the parser supports. These fields are:

- `content`: A string that contains the raw Markdown found in a specific location, based on the header metadata at that point in the document.
- `sections`: An object that contains subfields for the header metadata up to the desired header level. For example, when `markdownHeaderDepth` is set to `h3`, contains string fields `h1`, `h2`, and `h3`. These fields are indexed by mirroring this structure in the index, or through field mappings in the format `/sections/h1`, `sections/h2`, etc. For in-context examples, see the index and indexer configurations in the following samples. The subfields contained are:
 - `h1` - A string containing the `h1` header value. Empty string if not set at this point in the document.
 - (Optional) `h2` - A string containing the `h2` header value. Empty string if not set at this point in the document.
 - (Optional) `h3` - A string containing the `h3` header value. Empty string if not set at this point in the document.
 - (Optional) `h4` - A string containing the `h4` header value. Empty string if not set at this point in the document.
 - (Optional) `h5` - A string containing the `h5` header value. Empty string if not set at this point in the document.
 - (Optional) `h6` - A string containing the `h6` header value. Empty string if not set at this point in the document.

- `ordinal_position`: An integer value that indicates the position of the section within the document hierarchy. This field is used for ordering the sections in their original sequence as they appear in the document, beginning with an ordinal position of 1 and incrementing sequentially for each content block.

This implementation uses [field mappings](#) in the indexer to map from the enriched content to the index. For more information about the parsed one-to-many document structure, see [Index Markdown blobs](#).

This example provides samples of how to index data both with and without field mappings. In this case, we know that `h1` contains the title of the document, so we can map it to a field named `title`. We'll also be mapping the `h2` and `h3` fields to `h2_subheader` and `h3_subheader`, respectively. The `content` and `ordinal_position` fields require no mapping because they're extracted from the Markdown directly into fields using those names. For an example of a full index schema that doesn't require field mappings, see the end of this section.

HTTP

```
### Create an index
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "name": "sample-markdown-index",
  "fields": [
    {"name": "id", "type": "Edm.String", "key": true, "searchable": true,
    "retrievable": true, "filterable": true, "facetable": true, "sortable": true},
    {"name": "content", "type": "Edm.String", "key": false, "searchable": true,
    "retrievable": true, "filterable": true, "facetable": true, "sortable": true},
    {"name": "title", "type": "Edm.String", "searchable": true, "retrievable": true,
    "filterable": true, "facetable": true, "sortable": true},
    {"name": "h2_subheader", "type": "Edm.String", "searchable": true,
    "retrievable": true, "filterable": true, "facetable": true, "sortable": true},
    {"name": "h3_subheader", "type": "Edm.String", "searchable": true,
    "retrievable": true, "filterable": true, "facetable": true, "sortable": true},
    {"name": "ordinal_position", "type": "Edm.Int32", "searchable": false,
    "retrievable": true, "filterable": true, "facetable": true, "sortable": true}
  ]
}
```

Index schema in a configuration with no field mappings

Field mappings allow you to manipulate and filter enriched content to fit into your desired index shape. However, you might just want to take the enriched content directly. In that case, the schema would look like:

HTTP

```
{  
  "name": "sample-markdown-index",  
  "fields": [  
    {"name": "id", "type": "Edm.String", "key": true, "searchable": true,  
     "retrievable": true, "filterable": true, "facetable": true, "sortable": true},  
    {"name": "content", "type": "Edm.String", "key": false, "searchable": true,  
     "retrievable": true, "filterable": true, "facetable": true, "sortable": true},  
    {"name": "sections",  
      "type": "Edm.ComplexType",  
      "fields": [  
        {"name": "h1", "type": "Edm.String", "searchable": true, "retrievable":  
         true, "filterable": true, "facetable": true, "sortable": true},  
        {"name": "h2", "type": "Edm.String", "searchable": true, "retrievable":  
         true, "filterable": true, "facetable": true, "sortable": true},  
        {"name": "h3", "type": "Edm.String", "searchable": true, "retrievable":  
         true, "filterable": true, "facetable": true, "sortable": true}  
      ]  
    },  
    {"name": "ordinal_position", "type": "Edm.Int32", "searchable": false,  
     "retrievable": true, "filterable": true, "facetable": true, "sortable": true}  
  ]  
}
```

To reiterate, we have subfields up to `h3` in the sections object because `markdownHeaderDepth` is set to `h3`.

If you use this schema, be sure to adjust later requests accordingly. This will require removing the field mappings from the indexer configuration and updating search queries to use the corresponding field names.

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads and indexes data, and optionally provides a schedule to automate the data refresh.

HTTP

```
### Create and run an indexer  
POST {{baseUrl}}/indexers?api-version=2025-09-01 HTTP/1.1  
Content-Type: application/json  
api-key: {{apiKey}}  
  
{  
  "name": "sample-markdown-indexer",  
  "dataSourceName": "sample-markdown-ds",  
  "targetIndexName": "sample-markdown-index",
```

```

"parameters" : {
  "configuration": {
    "parsingMode": "markdown",
    "markdownParsingSubmode": "oneToMany",
    "markdownHeaderDepth": "h3"
  }
},
"fieldMappings" : [
  {
    "sourceFieldName": "/sections/h1",
    "targetFieldName": "title",
    "mappingFunction": null
  }
]
}

```

Key points:

- The indexer will only parse headers up to `h3`. Any lower-level headers (`h4, h5, h6`) are treated as plain text and show up in the `content` field. This is why the index and field mappings only exist up to a depth of `h3`.
- The `content` and `ordinal_position` fields require no field mapping because they exist with those names in the enriched content.

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```

### Query the index
POST {{baseUrl}}/indexes/sample-markdown-index/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "search": "*",
  "count": true
}

```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 6b94e605-55e8-47a5-ae15-834f926ddd14
elapsed-time: 77
Date: Fri, 25 Oct 2024 20:22:58 GMT
Connection: close

{
    "@odata.context": "https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes('sample-markdown-index')/$metadata#docs(*)",
    "@odata.count": 22,
    "value": [
        <22 search documents here>
    ]
}
```

Add a `search` parameter to search on a string.

HTTP

```
### Query the index
POST {{baseUrl}}/indexes/sample-markdown-index/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

{
    "search": "h4",
    "count": true
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
```

```

Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: ec5d03f1-e3e7-472f-9396-7ff8e3782105
elapsed-time: 52
Date: Fri, 25 Oct 2024 20:26:29 GMT
Connection: close

{
  "@odata.context": "https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes('sample-markdown-index')/$metadata#docs(*)",
  "@odata.count": 1,
  "value": [
    {
      "@search.score": 0.8744742,
      "section_id": "aHR0cHM6Ly9hcmpmZ2Fubmpma2ZpbGVzLmJsb2IuY29yZS53aW5kb3dzLm5ldC9tYXJrZG93bi10dXRvcmlhbC9zYW1wbGVfbWFya2Rvd24ubWQ7NA2",
      "content": "#### h4 example\r\n##### h5 example\r\n##### h6 example\r\nThis is an example of content underneath a header.\r\n",
      "title": "Project Documentation",
      "h2_subheader": "Headers",
      "h3_subheader": "h3 example",
      "ordinal_position": 4
    }
  ]
}

```

Key points:

- Because the `markdownHeaderDepth` is set to `h3`, the `h4`, `h5`, and `h6` headers are treated as plaintext, so they appear in the `content` field.
- Ordinal position here is `4`. This content appears fourth out of the 22 total content sections.

Add a `select` parameter to limit the results to fewer fields. Add a `filter` to further narrow the search.

HTTP

```

### Query the index
POST {{baseUrl}}/indexes/sample-markdown-index/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "search": "Markdown",
  "count": true,
  "select": "title, content, h2_subheader",
}

```

```
        "filter": "h2_subheader eq 'Conclusion'"  
    }
```

JSON

```
HTTP/1.1 200 OK  
Transfer-Encoding: chunked  
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;  
charset=utf-8  
Content-Encoding: gzip  
Vary: Accept-Encoding  
Server: Microsoft-IIS/10.0  
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains  
Preference-Applied: odata.include-annotations="*"  
OData-Version: 4.0  
request-id: a6f9bd46-a064-4e28-818f-ea077618014b  
elapsed-time: 35  
Date: Fri, 25 Oct 2024 20:36:10 GMT  
Connection: close  
  
{  
    "@odata.context": "https://<YOUR-SEARCH-SERVICE-  
NAME>.search.windows.net/indexes('sample-markdown-index')/$metadata#docs(*)",  
    "@odata.count": 1,  
    "value": [  
        {  
            "@search.score": 1.1029507,  
            "content": "Markdown is a lightweight yet powerful tool for writing  
documentation. It supports a variety of formatting options while maintaining  
simplicity and readability.\r\n\r\nThank you for reviewing this example!",  
            "title": "Project Documentation",  
            "h2_subheader": "Conclusion"  
        }  
    ]  
}
```

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case sensitive. For more information and examples, see [Create a query](#).

ⓘ Note

The `$filter` parameter only works on fields that were marked filterable at the creation of your index.

Reset and rerun

Indexers can be reset to clear execution history, which allows a full rerun. The following GET requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{baseUrl}}/indexers/sample-markdown-indexer/reset?api-version=2025-09-01
HTTP/1.1
api-key: {{apiKey}}

### Run the indexer
POST {{baseUrl}}/indexers/sample-markdown-indexer/run?api-version=2025-09-01
HTTP/1.1
api-key: {{apiKey}}

### Check indexer status
GET {{baseUrl}}/indexers/sample-markdown-indexer/status?api-version=2025-09-01
HTTP/1.1
api-key: {{apiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

Next steps

Now that you're familiar with the basics of Azure Blob indexing, take a closer look at indexer configuration for Markdown blobs in Azure Storage:

[Configure Markdown blob indexing](#)

Last updated on 11/21/2025

Tutorial: Index from multiple data sources using the .NET SDK

Azure AI Search supports importing, analyzing, and indexing data from multiple data sources into a single consolidated search index.

This C# tutorial uses the [Azure.Search.Documents](#) client library in the Azure SDK for .NET to index sample hotel data from an Azure Cosmos DB instance. You then merge the data with hotel room details drawn from Azure Blob Storage documents. The result is a combined hotel search index containing hotel documents, with rooms as complex data types.

In this tutorial, you:

- ✓ Upload sample data to data sources
- ✓ Identify the document key
- ✓ Define and create the index
- ✓ Index hotel data from Azure Cosmos DB
- ✓ Merge hotel room data from Blob Storage

Overview

This tutorial uses [Azure.Search.Documents](#) to create and run multiple indexers. You upload sample data to two Azure data sources and configure an indexer that pulls from both sources to populate a single search index. The two data sets must have a value in common to support the merge. In this tutorial, that field is an ID. As long as there's a field in common to support the mapping, an indexer can merge data from disparate resources: structured data from Azure SQL, unstructured data from Blob Storage, or any combination of [supported data sources](#) on Azure.

A finished version of the code in this tutorial can be found in the following project:

- [multiple-data-sources/v11 \(GitHub\)](#) ↗

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#) ↗ .
- An [Azure Cosmos DB for NoSQL account](#).
- An [Azure Storage account](#).
- An [Azure AI Search service](#).
- [Visual Studio](#) ↗ .

Note

You can use a free search service for this tutorial. The free tier limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before you start, make sure you have room on your service to accept the new resources.

Prepare services

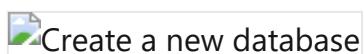
This tutorial uses Azure AI Search for indexing and queries, Azure Cosmos DB for the first data set, and Azure Blob Storage for the second data set.

If possible, create all services in the same region and resource group for proximity and manageability. In practice, your services can be in any region.

This sample uses two small sets of data describing seven fictional hotels. One set describes the hotels themselves and will be loaded into an Azure Cosmos DB database. The other set contains hotel room details and is provided as seven separate JSON files to be uploaded into Azure Blob Storage.

Start with Azure Cosmos DB

1. Sign in to the [Azure portal](#) and select your Azure Cosmos DB account.
2. From the left pane, select **Data Explorer**.
3. Select **New Container > New Database**.



4. Enter **hotel-rooms-db** for the name. Accept the default values for the remaining settings.



5. Create a container that targets the database you previously created. Enter **hotels** for the container name and **/HotelId** for the partition key.

New Container

* Database id ⓘ

Create new Use existing

hotel-rooms-db

* Container id ⓘ

hotels

* Partition key ⓘ

/HotelId

Provision dedicated throughput for this container ⓘ

Unique keys ⓘ

Comma separated paths e.g. /firstName,/address/z... 

+ Add unique key

Analytical store ⓘ

On Off

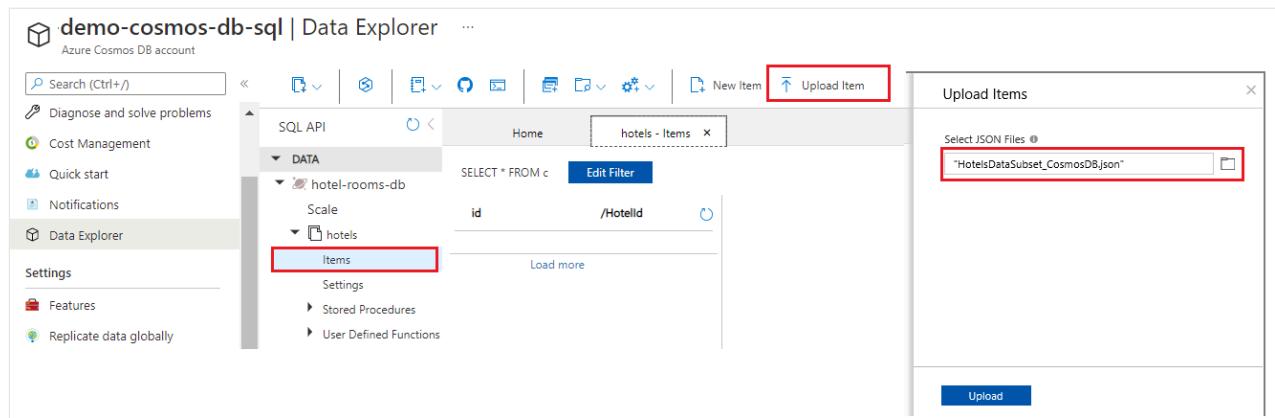
Azure Synapse Link is required for creating an analytical store container. Enable Synapse Link for this Cosmos DB account. [Learn more](#)

Enable

> Advanced

OK

6. Select **hotels** > **Items**, and then select **Upload Item** on the command bar.
7. Upload the JSON file from the `cosmosdb` folder in [multiple-data-sources/v11](#).



8. Use the refresh button to refresh your view of the items in the hotels collection. You should see seven new database documents listed.
9. From the left pane, select **Settings > Keys**.
10. Make a note of a connection string. You need this value for `appsettings.json` in a later step. If you didn't use the suggested **hotel-rooms-db** database name, copy the database name as well.

Azure Blob Storage

1. Sign in to the [Azure portal](#) and select your Azure Storage account.
2. From the left pane, select **Data storage > Containers**.
3. [Create a blob container](#) named **hotel-rooms** to store the sample hotel room JSON files. You can set the access level to any valid value.



4. Open the container, and then select **Upload** on the command bar.
5. Upload the seven JSON files from the `blob` folder in [multiple-data-sources/v11](#).



6. From the left pane, select **Security + networking > Access keys**.
7. Make a note of the account name and a connection string. You need both values for `appsettings.json` in a later step.

Azure AI Search

The third component is Azure AI Search, which you can [create in the Azure portal](#) or [find an existing search service](#) in your Azure resources.

Copy an admin key and URL for Azure AI Search

To authenticate to your search service, you need the service URL and an access key. Having a valid key establishes trust on a per-request basis between the application sending the request and the service handling it.

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Overview**.
3. Make a note of the URL, which should look like `https://my-service.search.windows.net`.
4. From the left pane, select **Settings > Keys**.
5. Make a note of an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests for adding, modifying, and deleting objects.

Set up your environment

1. Open the `AzureSearchMultipleDataSources.sln` file from [multiple-data-sources/v11](#) in Visual Studio.
2. In Solution Explorer, right-click the project and select **Manage NuGet Packages for Solution....**
3. On the **Browse** tab, find and install the following packages:
 - **Azure.Search.Documents** (version 11.0 or later)
 - **Microsoft.Extensions.Configuration**
 - **Microsoft.Extensions.Configuration.Json**
4. In Solution Explorer, edit the `appsettings.json` file with the connection information you collected in the previous steps.

JSON

```
{  
  "SearchServiceUri": "<YourSearchServiceURL>",  
  "SearchServiceAdminApiKey": "<YourSearchServiceAdminApiKey>",  
  "BlobStorageAccountName": "<YourBlobStorageAccountName>",  
  "BlobStorageConnectionString": "<YourBlobStorageConnectionString>",  
  "CosmosDBConnectionString": "<YourCosmosDBConnectionString>"}
```

```
        "CosmosDBDatabaseName": "hotel-rooms-db"  
    }
```

Map key fields

Merging content requires that both data streams are targeting the same documents in the search index.

In Azure AI Search, the key field uniquely identifies each document. Every search index must have exactly one key field of type `Edm.String`. That key field must be present for each document in a data source that is added to the index. (In fact, it's the only required field.)

When indexing data from multiple data sources, make sure each incoming row or document contains a common document key. This allows you to merge data from two physically distinct source documents into a new search document in the combined index.

It often requires some up-front planning to identify a meaningful document key for your index and to make sure it exists in both data sources. In this demo, the `HotelId` key for each hotel in Azure Cosmos DB is also present in the rooms JSON blobs in Blob Storage.

Azure AI Search indexers can use field mappings to rename and even reformat data fields during the indexing process, so that source data can be directed to the correct index field. For example, in Azure Cosmos DB, the hotel identifier is called `HotelId`, but in the JSON blob files for the hotel rooms, the hotel identifier is named `Id`. The program handles this discrepancy by mapping the `Id` field from the blobs to the `HotelId` key field in the indexer.

(!) Note

In most cases, autogenerated document keys, such as those created by default by some indexers, don't make good document keys for combined indexes. In general, use a meaningful, unique key value that already exists in your data sources or can be easily added.

Explore the code

When the data and configuration settings are in place, the sample program in `AzureSearchMultipleDataSources.sln` should be ready to build and run.

This simple C#/.NET console app performs the following tasks:

- Creates a new index based on the data structure of the C# Hotel class, which also references the Address and Room classes.
- Creates a new data source and an indexer that maps Azure Cosmos DB data to index fields. These are both objects in Azure AI Search.
- Runs the indexer to load hotel data from Azure Cosmos DB.
- Creates a second data source and an indexer that maps JSON blob data to index fields.
- Runs the second indexer to load hotel room data from Blob Storage.

Before you run the program, take a minute to study the code, index definition, and indexer definition. The relevant code is in two files:

- `Hotel.cs` contains the schema that defines the index.
- `Program.cs` contains functions that create the Azure AI Search index, data sources, and indexers, and load the combined results into the index.

Create an index

This sample program uses `CreateIndexAsync` to define and create an Azure AI Search index. It takes advantage of the `FieldBuilder` class to generate an index structure from a C# data model class.

The data model is defined by the Hotel class, which also contains references to the Address and Room classes. The FieldBuilder drills down through multiple class definitions to generate a complex data structure for the index. Metadata tags are used to define the attributes of each field, such as whether it's searchable or sortable.

The program deletes any existing index of the same name before creating the new one, in case you want to run this example more than once.

The following snippets from the `Hotel.cs` file show single fields, followed by a reference to another data model class, `Room[]`, which in turn is defined in `Room.cs` file (not shown).

C#

```
...
[SimpleField(IsFilterable = true, IsKey = true)]
public string HotelId { get; set; }

[SearchableField(IsFilterable = true, IsSortable = true)]
public string HotelName { get; set; }
...
public Room[] Rooms { get; set; }
...
```

In the `Program.cs` file, a `SearchIndex` is defined with a name and a field collection generated by the `FieldBuilder.Build` method, and then created as follows:

C#

```
private static async Task CreateIndexAsync(string indexName, SearchIndexClient indexClient)
{
    // Create a new search index structure that matches the properties of the Hotel
    // class.
    // The Address and Room classes are referenced from the Hotel class. The
    FieldBuilder
    // will enumerate these to create a complex data structure for the index.
    FieldBuilder builder = new FieldBuilder();
    var definition = new SearchIndex(indexName, builder.Build(typeof(Hotel)));

    await indexClient.CreateIndexAsync(definition);
}
```

Create Azure Cosmos DB data source and indexer

The main program includes logic to create the Azure Cosmos DB data source for the hotels data.

First, it concatenates the Azure Cosmos DB database name to the connection string. It then defines a `SearchIndexerDataSourceConnection` object.

C#

```
private static async Task CreateAndRunCosmosDbIndexerAsync(string indexName,
SearchIndexerClient indexerClient)
{
    // Append the database name to the connection string
    string cosmosConnectionString =
        configuration["CosmosDBConnectionString"]
        + ";Database="
        + configuration["CosmosDBDatabaseName"];

    SearchIndexerDataSourceConnection cosmosDbDataSource = new
    SearchIndexerDataSourceConnection(
        name: configuration["CosmosDBDatabaseName"],
        type: SearchIndexerDataSourceType.CosmosDb,
        connectionString: cosmosConnectionString,
        container: new SearchIndexerDataContainer("hotels"));

    // The Azure Cosmos DB data source does not need to be deleted if it already
    exists,
    // but the connection string might need to be updated if it has changed.
    await indexerClient.CreateOrUpdateDataSourceConnectionAsync(cosmosDbDataSource);
```

After the data source is created, the program sets up an Azure Cosmos DB indexer named `hotel-rooms-cosmos-indexer`.

The program updates any existing indexers with the same name, overwriting the existing indexer with the content of the previous code. It also includes reset and run actions, in case you want to run this example more than once.

The following example defines a schedule for the indexer, so that it runs once per day. You can remove the `schedule` property from this call if you don't want the indexer to automatically run again in the future.

C#

```
SearchIndexer cosmosDbIndexer = new SearchIndexer(
    name: "hotel-rooms-cosmos-indexer",
    dataSourceName: cosmosDbDataSource.Name,
    targetIndexName: indexName)
{
    Schedule = new IndexingSchedule(TimeSpan.FromDays(1))
};

// Indexers keep metadata about how much they have already indexed.
// If we already ran the indexer, it "remembers" and does not run again.
// To avoid this, reset the indexer if it exists.
try
{
    await indexerClient.GetIndexerAsync(cosmosDbIndexer.Name);
    // Reset the indexer if it exists.
    await indexerClient.ResetIndexerAsync(cosmosDbIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    // If the indexer does not exist, 404 will be thrown.
}

await indexerClient.CreateOrUpdateIndexerAsync(cosmosDbIndexer);

Console.WriteLine("Running Azure Cosmos DB indexer...\n");

try
{
    // Run the indexer.
    await indexerClient.RunIndexerAsync(cosmosDbIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 429)
{
    Console.WriteLine("Failed to run indexer: {0}", ex.Message);
}
```

This example includes a simple try-catch block to report any errors that might occur during execution.

After the Azure Cosmos DB indexer runs, the search index contains a full set of sample hotel documents. However, the rooms field for each hotel is an empty array, since the Azure Cosmos DB data source omits room details. Next, the program pulls from Blob Storage to load and merge the room data.

Create Blob Storage data source and indexer

To get the room details, the program first sets up a Blob Storage data source to reference a set of individual JSON blob files.

C#

```
private static async Task CreateAndRunBlobIndexerAsync(string indexName,
SearchIndexerClient indexerClient)
{
    SearchIndexerDataSourceConnection blobDataSource = new
    SearchIndexerDataSourceConnection(
        name: configuration["BlobStorageAccountName"],
        type: SearchIndexerDataSourceType.AzureBlob,
        connectionString: configuration["BlobStorageConnectionString"],
        container: new SearchIndexerDataContainer("hotel-rooms"));

    // The blob data source does not need to be deleted if it already exists,
    // but the connection string might need to be updated if it has changed.
    await indexerClient.CreateOrUpdateDataSourceConnectionAsync(blobDataSource);
```

After the data source is created, the program sets up a blob indexer named `hotel-rooms-blob-indexer`, as shown below.

The JSON blobs contain a key field named `Id` instead of `HotelId`. The code uses the `FieldMapping` class to tell the indexer to direct the `Id` field value to the `HotelId` document key in the index.

Blob Storage indexers can use `IndexingParameters` to specify a parsing mode. You should set different parsing modes depending on whether blobs represent a single document or multiple documents within the same blob. In this example, each blob represents a single JSON document, so the code uses the `json` parsing mode. For more information about indexer parsing parameters for JSON blobs, see [Index JSON blobs](#).

This example defines a schedule for the indexer, so that it runs once per day. You can remove the `schedule` property from this call if you don't want the indexer to automatically run again in the future.

C#

```

IndexingParameters parameters = new IndexingParameters();
parameters.Configuration.Add("parsingMode", "json");

SearchIndexer blobIndexer = new SearchIndexer(
    name: "hotel-rooms-blob-indexer",
    dataSourceName: blobDataSource.Name,
    targetIndexName: indexName)
{
    Parameters = parameters,
    Schedule = new IndexingSchedule(TimeSpan.FromDays(1))
};

// Map the Id field in the Room documents to the HotelId key field in the index
blobIndexer.FieldMappings.Add(new FieldMapping("Id") { TargetFieldName = "HotelId" });

// Reset the indexer if it already exists
try
{
    await indexerClient.GetIndexerAsync(blobIndexer.Name);
    await indexerClient.ResetIndexerAsync(blobIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 404) { }

await indexerClient.CreateOrUpdateIndexerAsync(blobIndexer);

try
{
    // Run the indexer.
    await searchService.Indexers.RunAsync(blobIndexer.Name);
}
catch (CloudException e) when (e.Response.StatusCode == (HttpStatusCode)429)
{
    Console.WriteLine("Failed to run indexer: {0}", e.Response.Content);
}

```

Because the index is already populated with hotel data from the Azure Cosmos DB database, the blob indexer updates the existing documents in the index and adds the room details.

Note

If you have the same non-key fields in both of your data sources, and the data in those fields doesn't match, the index contains the values from whichever indexer ran most recently. In our example, both data sources contain a `HotelName` field. If for some reason the data in this field is different, for documents with the same key value, the `HotelName` data from the most recently indexed data source is the value stored in the index.

Search

After you run the program, you can explore the populated search index using [Search explorer](#) in the Azure portal.

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Search management > Indexes**.
3. Select **hotel-rooms-sample** from the list of indexes.
4. On the **Search explorer** tab, enter a query for a term like `Luxury`.

You should see at least one document in the results. This document should contain a list of room objects in its `Rooms` array.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure AI Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code checks for existing objects and deletes or updates them so that you can rerun the program. You can also use the Azure portal to delete indexes, indexers, and data sources.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal using the All resources or Resource groups link in the left pane.

Next step

Now that you're familiar with ingesting data from multiple sources, take a closer look at indexer configuration, starting with Azure Cosmos DB:

[Configure an Azure Cosmos DB for NoSQL indexer](#)

Tutorial: Create a custom analyzer for phone numbers

In search solutions, strings that have complex patterns or special characters can be challenging to work with because the [default analyzer](#) strips out or misinterprets meaningful parts of a pattern. This results in a poor search experience where users can't find the information they expect. Phone numbers are a classic example of strings that are difficult to analyze. They come in various formats and include special characters that the default analyzer ignores.

With phone numbers as its subject, this tutorial uses the [Search Service REST APIs](#) to solve patterned data problems using a [custom analyzer](#). This approach can be used as is for phone numbers or adapted for fields with the same characteristics (patterned with special characters), such as URLs, emails, postal codes, and dates.

In this tutorial, you:

- ✓ Understand the problem
- ✓ Develop an initial custom analyzer for handling phone numbers
- ✓ Test the custom analyzer
- ✓ Iterate on custom analyzer design to further improve results

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#).
- [Visual Studio Code](#) with the [REST Client extension](#).

Download files

Source code for this tutorial is in the [custom-analyzer.rest](#) file in the [Azure-Samples/azure-search-rest-samples](#) GitHub repository.

Copy an admin key and URL

The REST calls in this tutorial require a search service endpoint and an admin API key. You can get these values from the Azure portal.

1. Sign in to the [Azure portal](#) and select your search service.

2. From the left pane, select **Overview** and copy the endpoint. It should be in this format:

```
https://my-service.search.windows.net
```

3. From the left pane, select **Settings > Keys** and copy an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests to add, modify, or delete objects.

The screenshot shows two side-by-side Azure search service pages. The top page is the 'Overview' section of a service named 'new-demo-search-svc'. The URL 'https://new-demo-search-svc.search.windows.net' is displayed in the top right. The bottom page is the 'Keys' section of the same service, showing two admin keys: 'Primary admin key' and 'Secondary admin key', both of which are highlighted with red boxes. Red numbers 1 and 2 are overlaid on the respective sections to indicate the steps described in the text above.

Create an initial index

1. Open a new text file in Visual Studio Code.

2. Set variables to the search endpoint and the API key you collected in the previous section.

```
HTTP
@baseUrl = PUT-YOUR-SEARCH-SERVICE-URL-HERE
@apiKey = PUT-YOUR-ADMIN-API-KEY-HERE
```

3. Save the file with a `.rest` file extension.

4. Paste the following example to create a small index called `phone-numbers-index` with two fields: `id` and `phone_number`.

```
HTTP
```

```

### Create a new index
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "name": "phone-numbers-index",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": true,
      "filterable": false,
      "facetable": false,
      "sortable": true
    },
    {
      "name": "phone_number",
      "type": "Edm.String",
      "sortable": false,
      "searchable": true,
      "filterable": false,
      "facetable": false
    }
  ]
}

```

You haven't defined an analyzer yet, so the `standard.lucene` analyzer is used by default.

5. Select **Send request**. You should have an `HTTP/1.1 201 Created` response, and the response body should include the JSON representation of the index schema.

6. Load data into the index using documents that contain various phone number formats. This is your test data.

HTTP

```

### Load documents
POST {{baseUrl}}/indexes/phone-numbers-index/docs/index?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "value": [
    {
      "@search.action": "upload",
      "id": "1",
      "phone_number": "425-555-0100"
    },
    {

```

```

        "@search.action": "upload",
        "id": "2",
        "phone_number": "(321) 555-0199"
    },
    {
        "@search.action": "upload",
        "id": "3",
        "phone_number": "+1 425-555-0100"
    },
    {
        "@search.action": "upload",
        "id": "4",
        "phone_number": "+1 (321) 555-0199"
    },
    {
        "@search.action": "upload",
        "id": "5",
        "phone_number": "4255550100"
    },
    {
        "@search.action": "upload",
        "id": "6",
        "phone_number": "13215550199"
    },
    {
        "@search.action": "upload",
        "id": "7",
        "phone_number": "425 555 0100"
    },
    {
        "@search.action": "upload",
        "id": "8",
        "phone_number": "321.555.0199"
    }
]
}

```

7. Try queries similar to what a user might type. For example, a user might search for (425) 555-0100 in any number of formats and still expect results to be returned. Start by searching (425) 555-0100.

HTTP

```

### Search for a phone number
POST {{baseUrl}}/indexes/phone-numbers-index/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
    "search": "(425) 555-0100"
}

```

The query returns three out of four expected results but also returns two unexpected results.

JSON

```
{  
  "value": [  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "+1 425-555-0100"  
    },  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "425 555 0100"  
    },  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "425-555-0100"  
    },  
    {  
      "@search.score": 0.020766128,  
      "phone_number": "(321) 555-0199"  
    },  
    {  
      "@search.score": 0.020766128,  
      "phone_number": "+1 (321) 555-0199"  
    }  
  ]  
}
```

8. Try again without any formatting: 4255550100.

HTTP

```
### Search for a phone number  
POST {{baseUrl}}/indexes/phone-numbers-index/docs/search?api-version=2025-09-01  
HTTP/1.1  
Content-Type: application/json  
api-key: {{apiKey}}  
  
{  
  "search": "4255550100"  
}
```

This query does even worse, returning only one of four correct matches.

JSON

```
{  
  "value": [  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "+1 425-555-0100"  
    },  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "425 555 0100"  
    },  
    {  
      "@search.score": 0.05634898,  
      "phone_number": "425-555-0100"  
    },  
    {  
      "@search.score": 0.020766128,  
      "phone_number": "(321) 555-0199"  
    },  
    {  
      "@search.score": 0.020766128,  
      "phone_number": "+1 (321) 555-0199"  
    }  
  ]  
}
```

```

        "@search.score": 0.6015292,
        "phone_number": "4255550100"
    }
]
}

```

If you find these results confusing, you're not alone. The next section explains why you're getting these results.

Review how analyzers work

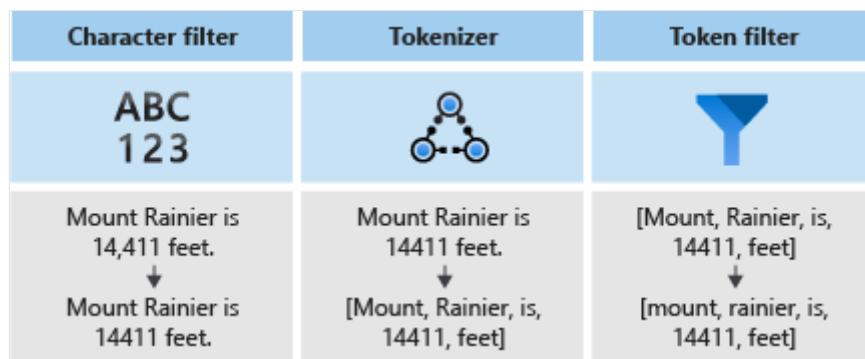
To understand these search results, you must understand what the analyzer is doing. From there, you can test the default analyzer using the [Analyze API](#), providing a foundation for designing an analyzer that better meets your needs.

An [analyzer](#) is a component of the [full-text search engine](#) responsible for processing text in query strings and indexed documents. Different analyzers manipulate text in different ways depending on the scenario. For this scenario, we need to build an analyzer tailored to phone numbers.

Analyzers consist of three components:

- [Character filters](#) that remove or replace individual characters from the input text.
- A [tokenizer](#) that breaks the input text into tokens, which become keys in the search index.
- [Token filters](#) that manipulate the tokens produced by the tokenizer.

The following diagram shows how these three components work together to tokenize a sentence.

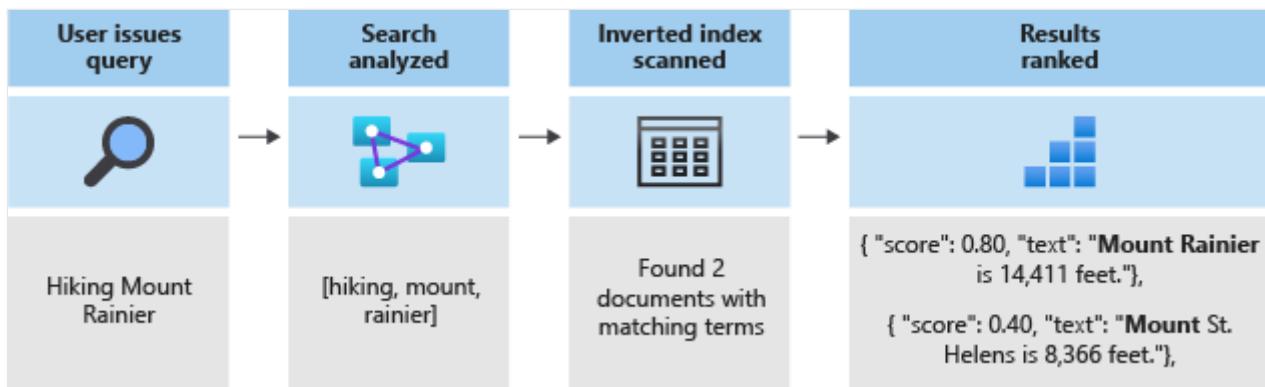


These tokens are then stored in an inverted index, which allows for fast, full-text searches. An inverted index enables full-text search by mapping all unique terms extracted during lexical analysis to the documents in which they occur. You can see an example in the following diagram:



All of search comes down to searching for the terms stored in the inverted index. When a user issues a query:

1. The query is parsed and the query terms are analyzed.
2. The inverted index is scanned for documents with matching terms.
3. The [scoring algorithm](#) ranks the retrieved documents.



If the query terms don't match the terms in your inverted index, results aren't returned. To learn more about how queries work, see [Full-text search in Azure AI Search](#).

Note

Partial term queries are an important exception to this rule. Unlike regular term queries, these queries (prefix query, wildcard query, and regex query) bypass the lexical analysis process. Partial terms are only lowercased before being matched against terms in the index. If an analyzer isn't configured to support these types of queries, you often receive unexpected results because matching terms don't exist in the index.

Test analyzers using the Analyze API

Azure AI Search provides an [Analyze API](#) that allows you to test analyzers to understand how they process text.

Call the Analyze API using the following request:

HTTP

```
### Test analyzer
POST {{baseUrl}}/indexes/phone-numbers-index/analyze?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
    "text": "(425) 555-0100",
```

```
        "analyzer": "standard.lucene"  
    }
```

The API returns the tokens extracted from the text, using the analyzer you specified. The standard Lucene analyzer splits the phone number into three separate tokens.

JSON

```
{  
    "tokens": [  
        {  
            "token": "425",  
            "startOffset": 1,  
            "endOffset": 4,  
            "position": 0  
        },  
        {  
            "token": "555",  
            "startOffset": 6,  
            "endOffset": 9,  
            "position": 1  
        },  
        {  
            "token": "0100",  
            "startOffset": 10,  
            "endOffset": 14,  
            "position": 2  
        }  
    ]  
}
```

Conversely, the phone number `4255550100` formatted without any punctuation is tokenized into a single token.

JSON

```
{  
    "text": "4255550100",  
    "analyzer": "standard.lucene"  
}
```

Response:

JSON

```
{  
    "tokens": [  
        {  
            "token": "4255550100",  
            "startOffset": 0,  
            "endOffset": 10,  
            "position": 0  
        }  
    ]  
}
```

```
        "endOffset": 10,  
        "position": 0  
    }  
]  
}
```

Keep in mind that both query terms and the indexed documents undergo analysis. Thinking back to the search results from the previous step, you can start to see why those results are returned.

In the first query, unexpected phone numbers are returned because one of their tokens, 555, matched one of the terms you searched. In the second query, only the one number is returned because it's the only record that has a token matching 4255550100.

Build a custom analyzer

Now that you understand the results you're seeing, build a custom analyzer to improve the tokenization logic.

The goal is to provide intuitive search against phone numbers no matter what format the query or indexed string is in. To achieve this outcome, specify a [character filter](#), a [tokenizer](#), and a [token filter](#).

Character filters

Character filters process text before it's fed into the tokenizer. Common uses of character filters are filtering out HTML elements and replacing special characters.

For phone numbers, you want to remove whitespace and special characters because not all phone number formats contain the same special characters and spaces.

JSON

```
"charFilters": [  
    {  
        "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",  
        "name": "phone_char_mapping",  
        "mappings": [  
            "-=>",  
            "(=>",  
            ")=>",  
            "+=>",  
            ".=>",  
            "\u0020=>"  
        ]  
    }]
```

```
    }  
]
```

The filter removes `-` `(` `)` `+` `.` and spaces from the input.

[Expand table](#)

Input	Output
<code>(321) 555-0199</code>	<code>3215550199</code>
<code>321.555.0199</code>	<code>3215550199</code>

Tokenizers

Tokenizers split text into tokens and discard some characters, such as punctuation, along the way. In many cases, the goal of tokenization is to split a sentence into individual words.

For this scenario, use a keyword tokenizer, `keyword_v2`, to capture the phone number as a single term. This isn't the only way to solve this problem, as explained in the [Alternate approaches](#) section.

Keyword tokenizers always output the same text they're given as a single term.

[Expand table](#)

Input	Output
<code>The dog swims.</code>	<code>[The dog swims.]</code>
<code>3215550199</code>	<code>[3215550199]</code>

Token filters

Token filters modify or filter out the tokens generated by the tokenizer. One common use of a token filter is to lowercase all characters using a lowercase token filter. Another common use is filtering out [stopwords](#), such as `the`, `and`, or `is`.

While you don't need to use either of those filters for this scenario, use an nGram token filter to allow for partial searches of phone numbers.

JSON

```
"tokenFilters": [  
  {
```

```
"@odata.type": "#Microsoft.Azure.Search.NGramTokenFilterV2",
"name": "custom_ngram_filter",
"minGram": 3,
"maxGram": 20
}
]
```

NGramTokenFilterV2

The [nGram_v2 token filter](#) splits tokens into n-grams of a given size based on the `minGram` and `maxGram` parameters.

For the phone analyzer, `minGram` is set to `3` because that's the shortest substring users are expected to search. `maxGram` is set to `20` to ensure that all phone numbers, even with extensions, fit into a single n-gram.

The unfortunate side effect of n-grams is that some false positives are returned. You fix this in a later step by building out a separate analyzer for searches that doesn't include the n-gram token filter.

[] Expand table

Input	Output
[12345]	[123, 1234, 12345, 234, 2345, 345]
[3215550199]	[321, 3215, 32155, 321555, 3215550, 32155501, 321555019, 3215550199, 215, 2155, 21555, 215550, ...]

Analyzer

With the character filters, tokenizer, and token filters in place, you're ready to define the analyzer.

JSON

```
"analyzers": [
{
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "phone_analyzer",
    "tokenizer": "keyword_v2",
    "tokenFilters": [
        "custom_ngram_filter"
    ],
    "charFilters": [
        "phone_char_mapping"
    ]
}]
```

```
    ]  
}  
]
```

From the Analyze API, given the following inputs, outputs from the custom analyzer are as follows:

 Expand table

Input	Output
12345	[123, 1234, 12345, 234, 2345, 345]
(321) 555-0199	[321, 3215, 32155, 321555, 3215550, 32155501, 321555019, 3215550199, 215, 2155, 21555, 215550, ...]

All of the tokens in the output column exist in the index. If your query includes any of those terms, the phone number is returned.

Rebuild using the new analyzer

1. Delete the current index.

HTTP

```
### Delete the index  
DELETE {{baseUrl}}/indexes/phone-numbers-index?api-version=2025-09-01 HTTP/1.1  
api-key: {{apiKey}}
```

2. Recreate the index using the new analyzer. This index schema adds a custom analyzer definition and a custom analyzer assignment on the phone number field.

HTTP

```
### Create a new index  
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1  
Content-Type: application/json  
api-key: {{apiKey}}  
  
{  
  "name": "phone-numbers-index-2",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true,  
      "searchable": true,  
      "filterable": false,
```

```

    "facetable": false,
    "sortable": true
  },
  {
    "name": "phone_number",
    "type": "Edm.String",
    "sortable": false,
    "searchable": true,
    "filterable": false,
    "facetable": false,
    "analyzer": "phone_analyzer"
  }
],
"analyzers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "phone_analyzer",
    "tokenizer": "keyword_v2",
    "tokenFilters": [
      "custom_ngram_filter"
    ],
    "charFilters": [
      "phone_char_mapping"
    ]
  }
],
"charFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
    "name": "phone_char_mapping",
    "mappings": [
      "-=>",
      "(=>",
      ")=>",
      "+=>",
      ".=>",
      "\u0020=>"
    ]
  }
],
"tokenFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.NGramTokenFilterV2",
    "name": "custom_ngram_filter",
    "minGram": 3,
    "maxGram": 20
  }
]
}

```

Test the custom analyzer

After you recreate the index, test the analyzer using the following request:

HTTP

```
### Test custom analyzer
POST {{baseUrl}}/indexes/phone-numbers-index-2/analyze?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
  "text": "+1 (321) 555-0199",
  "analyzer": "phone_analyzer"
}
```

You should now see the collection of tokens resulting from the phone number.

JSON

```
{
  "tokens": [
    {
      "token": "132",
      "startOffset": 1,
      "endOffset": 17,
      "position": 0
    },
    {
      "token": "1321",
      "startOffset": 1,
      "endOffset": 17,
      "position": 0
    },
    {
      "token": "13215",
      "startOffset": 1,
      "endOffset": 17,
      "position": 0
    },
    ...
    ...
    ...
  ]
}
```

Revise the custom analyzer to handle false positives

After using the custom analyzer to make sample queries against the index, you should see that recall has improved and all matching phone numbers are now returned. However, the n-gram

token filter also causes some false positives to be returned. This is a common side effect of an n-gram token filter.

To prevent false positives, create a separate analyzer for querying. This analyzer is identical to the previous one, except that it omits the `custom_ngram_filter`.

JSON

```
{  
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",  
    "name": "phone_analyzer_search",  
    "tokenizer": "custom_tokenizer_phone",  
    "tokenFilters": [],  
    "charFilters": [  
        "phone_char_mapping"  
    ]  
}
```

In the index definition, specify both an `indexAnalyzer` and a `searchAnalyzer`.

JSON

```
{  
    "name": "phone_number",  
    "type": "Edm.String",  
    "sortable": false,  
    "searchable": true,  
    "filterable": false,  
    "facetable": false,  
    "indexAnalyzer": "phone_analyzer",  
    "searchAnalyzer": "phone_analyzer_search"  
}
```

With this change, you're all set. Here are your next steps:

1. Delete the index.
2. Recreate the index after you add the new custom analyzer (`phone_analyzer-search`) and assign that analyzer to the `phone-number` field's `searchAnalyzer` property.
3. Reload the data.
4. Retest the queries to verify that the search works as expected. If you're using the sample file, this step creates the third index named `phone-number-index-3`.

Alternate approaches

The analyzer described in the previous section is designed to maximize the flexibility for search. However, it does so at the cost of storing many potentially unimportant terms in the index.

The following example shows an alternative analyzer that's more efficient in tokenization, but it has drawbacks.

Given an input of `14255550100`, the analyzer can't logically chunk the phone number. For example, it can't separate the country code, `1`, from the area code, `425`. This discrepancy leads to the phone number not being returned if a user doesn't include a country code in their search.

JSON

```
"analyzers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "phone_analyzer_shingles",
    "tokenizer": "custom_tokenizer_phone",
    "tokenFilters": [
      "custom_shingle_filter"
    ]
  }
],
"tokenizers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.StandardTokenizerV2",
    "name": "custom_tokenizer_phone",
    "maxTokenLength": 4
  }
],
"tokenFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.ShingleTokenFilter",
    "name": "custom_shingle_filter",
    "minShingleSize": 2,
    "maxShingleSize": 6,
    "tokenSeparator": ""
  }
]
```

In the following example, the phone number is split into the chunks you normally expect a user to be search for.

 Expand table

Input	Output
(321) 555-0199	[321, 555, 0199, 321555, 5550199, 3215550199]

Depending on your requirements, this might be a more efficient approach to the problem.

Takeaways

This tutorial demonstrated the process of building and testing a custom analyzer. You created an index, indexed data, and then queried against the index to see what search results were returned. From there, you used the Analyze API to see the lexical analysis process in action.

While the analyzer defined in this tutorial offers an easy solution for searching against phone numbers, this same process can be used to build a custom analyzer for any scenario that shares similar characteristics.

Clean up resources

When you're working in your own subscription, it's a good idea to remove the resources that you no longer need at the end of a project. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you know how to create a custom analyzer, take a look at all of the different filters, tokenizers, and analyzers available for building a rich search experience:

[Custom analyzers in Azure AI Search](#)

Last updated on 11/21/2025

Tutorial: Index and enrich encrypted blobs for full-text search in Azure AI Search

10/09/2025

Learn how to use [Azure AI Search](#) to index documents that were encrypted with a customer-managed key in [Azure Blob Storage](#).

Normally, an indexer can't extract content from blobs that were encrypted using [client-side encryption](#) in the Azure Blob Storage client library. This is because the indexer doesn't have access to the customer-managed encryption key in [Azure Key Vault](#). However, using the [DecryptBlobFile custom skill](#) and the [Document Extraction skill](#), you can provide controlled access to the key to decrypt the files and then extract content from them. This unlocks the ability to index and enrich these documents without compromising the encryption status of your stored documents.

Starting with previously encrypted whole documents (unstructured text) such as PDF, HTML, DOCX, and PPTX in Azure Blob Storage, this tutorial uses a REST client and the Search REST APIs to:

- ✓ Define a pipeline that decrypts the documents and extracts text from them
- ✓ Define an index to store the output
- ✓ Execute the pipeline to create and load the index
- ✓ Explore results using full-text search and a rich query syntax

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure AI Search](#) on any tier or region.
- [Azure Storage](#), Standard performance (general-purpose v2).
- Blobs encrypted with a customer-managed key. To create sample data, see [Tutorial: Encrypt and decrypt blobs using Azure Key Vault](#).
- [Azure Key Vault](#) in the same subscription as Azure AI Search. The key vault must have [soft-delete](#) and [purge protection](#) enabled.

Custom skill deployment creates an Azure Function app and an Azure Storage account. These resources are created for you, so they aren't listed as a prerequisite. When you finish this tutorial, remember to clean up the resources so that you aren't billed for services you're not using.

(!) Note

Skillsets often require [attaching an Azure AI services multi-service resource](#). As written, this skillset has no dependency on Azure AI services, so no key is required. If you later add enrichments that invoke built-in skills, remember to update your skillset accordingly.

Deploy the custom skill

This tutorial uses the sample [DecryptBlobFile](#) project from the [Azure Search Power Skills](#) GitHub repository. In this section, you deploy the skill to an Azure Function so that it can be used in a skillset. A built-in deployment script creates an Azure Function resource with a **psdbf-function-app-** prefix and loads the skill. You're prompted to provide a subscription and resource group. Be sure to choose the subscription that contains your Azure Key Vault instance.

Operationally, the DecryptBlobFile skill takes the URL and SAS token for each blob as inputs. It outputs the downloaded, decrypted file using the file reference contract that Azure AI Search expects. Recall that DecryptBlobFile needs the encryption key to perform the decryption. As part of setup, you also create an access policy that grants DecryptBlobFile function access to the encryption key in Azure Key Vault.

1. On the [DecryptBlobFile landing page](#), select **Deploy to Azure** to open the Resource Manager template in the Azure portal.
2. Choose the subscription where your Azure Key Vault instance exists. This tutorial doesn't work if you choose a different subscription.
3. Select an existing resource group or create a new one. A dedicated resource group makes cleanup easier later.
4. Select **Review + create**, agree to the terms, and then select **Create** to deploy the Azure Function.

Home >

Custom deployment

Deploy from a custom template

Basics Review + create

Template

Customized template 2 resources

Edit template Edit parameters Visualize

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * <Choose a subscription that contains Azure Key Vault>

Resource group * (New) demo-group Create new

Instance details

Region * West Central US

Resource Prefix psdbf

Storage Account Type Standard_LRS

Review + create < Previous Next : Review + create >

5. Wait for the deployment to finish.

You should have an Azure Function app that contains the decryption logic and an Azure Storage resource that will store application data. In the next steps, you give the app permissions to access the key vault and collect information that you'll need for the REST calls.

Grant permissions in Azure Key Vault

1. Go to your Azure Key Vault service in the Azure portal. [Create an access policy](#) in the Azure Key Vault that grants key access to the custom skill.
2. From the left pane, select **Access policies**, and then select **+ Create** to start the [Create an access policy](#) wizard.

The screenshot shows the Microsoft Azure Key Vault interface. In the left sidebar, under 'my-demo-key-vault', the 'Access policies' option is selected and highlighted with a red box. At the top right, there is a 'Create' button, which is also highlighted with a red box. The main content area displays a table of access policies. The first row shows a user named 'Jane Doe' with the email 'jadoe@mycompany.co'. The permissions listed are 'Get, List, Update, Create, Import, D...' and 'Get, List, Set, Delete, Recover, Back...'. There are also tabs for 'Secret Permissions' and 'Certificate Permissions'.

3. On the **Permissions** page, under **Configure from template**, select **Azure Data Lake Storage or Azure Storage**.

4. Select **Next**.

5. On the **Principal** page, select the Azure Function instance you deployed. You can search for it using its resource prefix, which has a default value of **psdbf-function-app**.

6. Select **Next**.

7. On **Review + create**, select **Create**.

Collect app information

1. Go to the **psdbf-function-app** function in the Azure portal. Make a note of the following properties you'll need for the REST calls.

2. Get the function URL, which can be found under **Essentials** on the main page for the function.

The screenshot shows the Azure Function App details page for 'psdbf-function-app- 00001111-aaaa'. The 'Essentials' section is expanded. The 'URL' field is highlighted with a red box and contains the value '<https://psdbf-function-app.azurewebsites.net>'. Other visible properties include Resource group (encryption-demo), Status (Running), Location (West Central US), Subscription (Applied AI Docs Team), Subscription ID, Tags, Operating System (Windows), App Service Plan (WestCentralUSPlan (Y1:0)), Properties (See More), and Runtime version (3.3.1.0).

3. Get the host key code, which can be found by going to **App keys** and showing the **default** key, and copying the value.

The screenshot shows the Azure portal interface for a Function App named "psdbf-function-app-0000000000000000". The left sidebar has sections for Security, Events (preview), Functions (with "Functions" and "App keys" selected), App files, Proxies, Deployment (Deployment slots, Deployment Center), Settings (Configuration), and Configuration. The main content area is titled "Host keys (all functions)" and contains instructions about using host keys for HTTP functions. It features a "New host key" button, a "Show values" link, and a "Filter host keys" input field. A table lists host keys: "_master" and "default". Both rows have "Value" columns with "Hidden value. Click to show value" and "Renew key value" links. The "default" row is highlighted with a red box. Below this is a section titled "System keys" with a note about automatic management by the Function runtime.

Get an admin key and URL for Azure AI Search

1. Sign in to the [Azure portal](#).
2. On your search service **Overview** page, get the name of your search service. You can confirm your service name by reviewing the endpoint URL. For example, if your endpoint URL is `https://mydemo.search.windows.net`, your service name is `mydemo`.
3. In **Settings > Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

An API key is required in the header of every request sent to your service. A valid key establishes trust, on a per-request basis, between the application sending the request and the service that handles it.

Set up a REST client

Create the following variables for endpoints and keys.

 Expand table

Variable	Where to get it
admin-key	On the Keys page of the Azure AI Search service.
search-service-name	The name of the Azure AI Search service. The URL is <code>https://{{search-service-name}}.search.windows.net</code> .
storage-connection-string	In the storage account, on the Access Keys tab, select key1 > Connection string .
storage-container-name	The name of the blob container that has the encrypted files to be indexed.
function-uri	In the Azure Function, under Essentials on the main page.
function-code	In the Azure Function, by going to App keys , showing the default key, and copying the value.
api-version	Leave as 2020-06-30 .
datasource-name	Leave as encrypted-blobs-ds .
index-name	Leave as encrypted-blobs-idx .
skillset-name	Leave as encrypted-blobs-ss .
indexer-name	Leave as encrypted-blobs-ixr .

Review and run each request

Use the following HTTP requests to create the objects of an enrichment pipeline.

- **PUT request to create the index:** This search index holds the data that Azure AI Search uses and returns.
- **POST request to create the data source:** This data source specifies the connection to your storage account containing the encrypted blob files.
- **PUT request to create the skillset:** The skillset specifies the custom skill definition for the Azure Function that will decrypt the blob file data. It also specifies a [DocumentExtractionSkill](#) to extract the text from each document after it's decrypted.
- **PUT request to create the indexer:** Running the indexer retrieves the blobs, applies the skillset, and indexes and stores the results. You must run this request last. The custom skill in the skillset invokes the decryption logic.

Monitor indexing

Indexing and enrichment commence as soon as you submit the Create Indexer request. Depending on how many documents are in your storage account, indexing can take a while. To find out whether the indexer is still running, send a **Get Indexer Status** request and review the response to learn whether the indexer is running or view error and warning information.

If you're using the Free tier, expect the following message: "Could not extract content or metadata from your document. Truncated extracted text to '32768' characters". This message appears because blob indexing on the Free tier has a [32,000 limit on character extraction](#). You don't see this message for this data set on higher tiers.

Search your content

After indexer execution is finished, you can run queries to verify that the data is successfully decrypted and indexed. Go to your Azure AI Search service in the Azure portal and use the [Search Explorer](#) to run queries over the indexed data.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you've indexed encrypted files, you can [iterate on this pipeline by adding more skills](#) to enrich and gain more insights into your data.

If you're working with doubly encrypted data, you might want to investigate the index encryption features available in Azure AI Search. Although the indexer needs decrypted data for indexing purposes, once the index exists, it can be encrypted in a search index using a customer-managed key. This ensures that your data is always encrypted when at rest. For more information, see [Configure customer-managed keys for data encryption in Azure AI Search](#).

Tutorial: Index permission metadata from ADLS Gen2 and query with permission-filtered results

09/03/2025

This tutorial demonstrates how to index Azure Data Lake Storage (ADLS) Gen2 [Access Control Lists \(ACLs\)](#) and [role-based access control \(RBAC\)](#) scope into a search index using an indexer.

It also shows you how to structure a query that respects user access permissions. A successful query outcome confirms the permission transfer that occurred during index.

For more information about indexing ACLs, see [Use an ADLS Gen2 indexer to ingest permission metadata](#).

In this tutorial, you learn how to:

- ✓ Configure RBAC scope and ACLs on an `adlsgen2` data source
- ✓ Create an Azure AI Search index containing permission information fields
- ✓ Create and run an indexer to ingest permission information into an index from a data source
- ✓ Search the index you just created

Use a REST client to complete this tutorial and the [latest preview REST API](#). Currently, there's no support for ACL indexing in the Azure portal.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#) .
- Microsoft Entra ID authentication and authorization. Services and apps must be in the same tenant. Role assignments are used for each authenticated connection. Users and groups must be in the same tenant. You should have user and groups to work with. Creating tenants and security principals is out-of-scope for this tutorial.
- [ADLS Gen2](#) with a hierarchical namespace.
- Files in a hierarchical folder structure. This tutorial assumes the ADLS Gen2 demo of folder structure for file [/Oregon/Portland/Data.txt](#). This tutorial guides you through ACL assignment on folders and files so that you can complete the exercise successfully.
- [Azure AI Search](#), any region. Basic tier or higher is required for managed identity support.

- Visual Studio Code [with a REST client](#) or a [Python client](#) and [Jupyter package](#).

Prepare sample data

Upload the [state parks sample data](#) to a container in ADLS Gen2. The container name should be "parks" and it should have two folders: "Oregon" and "Washington".

Check search service configuration

You search service must be configured for Microsoft Entra ID authentication and authorization. Review this checklist to make sure you're prepared.

- [Enable role-based access](#)
- [Configure a system-assigned managed identity.](#)

Get a personal identity token for local testing

This tutorial assumes a REST client on a local system, connecting to Azure over a public internet connection.

[Follow these steps](#) to acquire a personal identity token and set up Visual Studio Code for local connections to your Azure resources.

Set permissions in ADLS Gen2

As a best practice, use [Group sets](#) rather than directly assigning [User](#) sets.

1. Grant the search service identity read access to the container. The indexer connects to Azure Storage under the search service identity. The search service must have **Storage Blob Data Reader** permissions to retrieve data.
2. Grant per-group or user permissions in the file hierarchy. In the file hierarchy, identify all [Group](#) and [User](#) sets that are assigned to containers, directories, and files.
3. You can use the Azure portal to manage ACLs. In Storage Browser, select the Oregon directory and then select **Manage ACL** from the context menu.
4. Add new security principals for users and groups.
5. Remove existing principals for owning groups, owning users, and other. These principals aren't supported for ACL indexing during the public preview.

Create a search index for permission metadata

Create an index that contains fields for content and [permission metadata](#).

Be sure to use the [latest preview REST API](#) or a prerelease Azure SDK that provides equivalent functionality. The permission filter properties are only available in the preview APIs.

For demo purposes, the permission field has `retrievable` enabled so that you can check the values from the index. In a production environment, you should disable `retrievable` to avoid leaking sensitive information.

JSON

```
{
  "name" : "my-adlsgen2-acl-index",
  "fields": [
    {
      "name": "name", "type": "Edm.String",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "description", "type": "Edm.String",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "location", "type": "Edm.String",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "state", "type": "Edm.String",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "AzureSearch_DocumentKey", "type": "Edm.String",
      "searchable": true, "filterable": false, "retrievable": true
      "stored": true,
      "key": true
    },
    {
      "name": "UserIds", "type": "Collection(Edm.String)",
      "permissionFilter": "userIds",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "GroupIds", "type": "Collection(Edm.String)",
      "permissionFilter": "groupIds",
      "searchable": true, "filterable": false, "retrievable": true
    },
    {
      "name": "RbacScope", "type": "Edm.String",
      "permissionFilter": "rbacScope",
      "searchable": true, "filterable": false, "retrievable": true
    }
  ]
}
```

```
        },
    ],
    "permissionFilterOption": "enabled"
}
```

Create a data source

Modify [data source configuration](#) to specify indexer permission ingestion and the types of permission metadata that you want to index.

A data source needs `indexerPermissionOptions`.

In this tutorial, use a system-assigned managed identity for the authenticated connection.

JSON

```
{
    "name" : "my-adlsgen2-acl-datasource",
    "type": "adlsgen2",
    "indexerPermissionOptions": ["userIds", "groupIds", "rbacScope"],
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>;"
    },
    "container": {
        "name": "parks",
        "query": null
    }
}
```

Create and run the indexer

Indexer configuration for permission ingestion is primarily about defining `fieldMappings` from [permission metadata](#).

JSON

```
{
    "name" : "my-adlsgen2-acl-indexer",
    "dataSourceName" : "my-adlsgen2-acl-datasource",
    "targetIndexName" : "my-adlsgen2-acl-index",
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "configuration": {
            "storageAccountName": "mystorageaccount",
            "storageContainerName": "parks"
        }
    }
}
```

```
        "dataToExtract": "contentAndMetadata",
        "parsingMode": "delimitedText",
        "firstLineContainsHeaders": true,
        "delimitedTextDelimiter": ",",
        "delimitedTextHeaders": ""
    },
    "fieldMappings": [
        { "sourceFieldName": "metadata_user_ids", "targetFieldName": "UserIds" },
        { "sourceFieldName": "metadata_group_ids", "targetFieldName": "GroupIds" },
        { "sourceFieldName": "metadata_rbac_scope", "targetFieldName": "RbacScope" }
    ]
}
```

After indexer creation and immediate run, the file content along with permission metadata information are indexed into the index.

Run a query to check results

Now that documents are loaded, you can issue queries against them by using [Documents - Search Post \(REST\)](#).

The URI is extended to include a query input, which is specified by using the `/docs/search` operator. The query token is passed in the request header. For more information, see [Query-Time ACL and RBAC enforcement](#).

HTTP

```
POST {{endpoint}}/indexes/stateparks/docs/search?api-version=2025-08-01-preview
Authorization: Bearer {{search-token}}
x-ms-query-source-authorization: {{search-token}}
Content-Type: application/json

{
    "search": "*",
    "select": "name,description,location,GroupIds",
    "orderby": "name asc"
}
```

Related content

- <https://github.com/Azure-Samples/azure-search-rest-samples/tree/main/Quickstart-ACL>

Tutorial: Vectorize images and text

Azure AI Search can extract and index both text and images from PDF documents stored in Azure Blob Storage. This tutorial shows you how to build a multimodal indexing pipeline in Azure AI Search that *chunks data using the built-in Text Split skill* and *uses multimodal embeddings* to vectorize text and images from the same document. Cropped images are stored in a knowledge store, and both text and visual content are vectorized and ingested in a searchable index.

In this tutorial, you use:

- A 36-page PDF document that combines rich visual content, such as charts, infographics, and scanned pages, with traditional text.
- An indexer and skillset to create an indexing pipeline that includes AI enrichment through skills.
- The [Document Extraction skill](#) for extracting normalized images and text. The [Text Split skill](#) chunks the data.
- The [Azure Vision multimodal embeddings skill](#) to vectorize text and images.
- A search index configured to store extracted text and image content. Some content is vectorized for vector-based similarity search.

This tutorial demonstrates a lower-cost approach for indexing multimodal content using the Document Extraction skill. It enables extraction and search over both text and images from documents pulled from Azure Blob Storage. However, it doesn't include locational metadata for text, such as page numbers or bounding regions. For a more comprehensive solution that includes structured text layout and spatial metadata, see [Tutorial: Vectorize from a structured document layout](#).

(!) Note

Image extraction by the Document Extraction skill isn't free. Setting `imageAction` to `generateNormalizedImages` in the skillset triggers image extraction, which is an extra charge. For billing information, see [Azure AI Search pricing](#).

Prerequisites

- [Microsoft Foundry resource](#). This resource provides access to the Azure Vision multimodal embedding model used in this tutorial. You must use a Foundry resource for skillset

access to this resource.

- [Azure AI Search](#). Configure your search service for role-based access control and a managed identity for connections to Azure Storage and Azure Vision. Your service must be on the Basic tier or higher. This tutorial isn't supported on the Free tier.
- [Azure Storage](#), used for storing sample data and for creating a [knowledge store](#).
- [Visual Studio Code](#) with a [REST client](#).

Limitations

- The [Azure Vision multimodal embeddings skill](#) has limited regional availability. When you create a Foundry resource, choose a region that provides multimodal embeddings. For an updated list of regions that provide multimodal embeddings, see the [Azure Vision documentation](#).

Prepare data

The following instructions apply to Azure Storage which provides the sample data and also hosts the knowledge store. A search service identity needs read access to Azure Storage to retrieve the sample data, and it needs write access to create the knowledge store. The search service creates the container for cropped images during skillset processing, using the name you provide in an environment variable.

1. Download the following sample PDF: [sustainable-ai-pdf](#)
2. In Azure Storage, create a new container named **sustainable-ai-pdf**.
3. [Upload the sample data file](#).
4. [Create role assignments and specify a managed identity in a connection string](#):
 - a. Assign **Storage Blob Data Reader** for data retrieval by the indexer. Assign **Storage Blob Data Contributor** and **Storage Table Data Contributor** to create and load the knowledge store. You can use either a system-assigned managed identity or a user-assigned managed identity for your search service role assignment.
 - b. For connections made using a system-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/;"  
}
```

- c. For connections made using a user-assigned managed identity, get a connection string that contains a ResourceId, with no account key or password. The ResourceId must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. Provide an identity using the syntax shown in the following example. Set userAssignedIdentity to the user-assigned managed identity. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/;"  
},  
"identity" : {  
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",  
    "userAssignedIdentity" : "/subscriptions/00000000-0000-0000-  
00000000/resourcegroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.ManagedIdentity/userAssignedIdentities/MY-DEMO-  
USER-MANAGED-IDENTITY"  
}
```

Prepare models

This tutorial assumes you have an existing Foundry resource through which the skill calls the Azure Vision multimodal 4.0 embedding model. The search service connects to the model during skillset processing using its managed identity. This section gives you guidance and links for assigning roles for authorized access.

1. Sign in to the Azure portal (not the Foundry portal) and find the Foundry resource. Make sure it's in a region that provides the [multimodal 4.0 API](#).
2. Select **Access control (IAM)**.
3. Select **Add** and then **Add role assignment**.
4. Search for **Cognitive Services User** and then select it.
5. Choose **Managed identity** and then assign your [search service managed identity](#).

Set up your REST file

For this tutorial, your local REST client connection to Azure AI Search requires an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Connect to a search service](#).

For authenticated connections that occur during indexer and skillset processing, the search service uses the role assignments you previously defined.

1. Start Visual Studio Code and create a new file.
2. Provide values for variables used in the request. For `@storageConnection`, make sure your connection string doesn't have a trailing semicolon or quotation marks. For `@imageProjectionContainer`, provide a container name that's unique in blob storage. Azure AI Search creates this container for you during skills processing.

HTTP

```
@searchUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE  
@searchApiKey = PUT-YOUR-ADMIN-API-KEY-HERE  
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE  
@cognitiveServicesUrl = PUT-YOUR-AZURE-AI-FOUNDRY-ENDPOINT-HERE  
@modelVersion = 2023-04-15  
@imageProjectionContainer=sustainable-ai-pdf-images
```

3. Save the file using a `.rest` or `.http` file extension. For help with the REST client, see [Quickstart: Full-text search using REST](#).

To get the Azure AI Search endpoint and API key:

1. Sign in to the [Azure portal](#), navigate to the search service **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

Create a data source

[Create Data Source \(REST\)](#) creates a data source connection that specifies what data to index.

HTTP

```
POST {{searchUrl}}/datasources?api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}
```

```
{
  "name": "doc-extraction-multimodal-embedding-ds",
  "description": null,
  "type": "azureblob",
  "subtype": null,
  "credentials": {
    "connectionString": "{{storageConnection}}"
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Location: https://<YOUR-SEARCH-SERVICE-NAME>.search.windows-
int.net:443/datasources('doc-extraction-multimodal-embedding-ds')?api-version=2025-
11-01-preview -Preview
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 4eb8bcc3-27b5-44af-834e-295ed078e8ed
elapsed-time: 346
Date: Sat, 26 Apr 2025 21:25:24 GMT
Connection: close

{
  "name": "doc-extraction-multimodal-embedding-ds",
  "description": null,
  "type": "azureblob",
  "subtype": null,
  "indexerPermissionOptions": [],
  "credentials": {
    "connectionString": null
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the parameters and their attributes.

For nested JSON, the index fields must be identical to the source fields. Currently, Azure AI Search doesn't support field mappings to nested JSON, so field names and data types must match completely. The following index aligns to the JSON elements in the raw content.

HTTP

```
### Create an index
POST {{searchUrl}}/indexes?api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
```

```
api-key: {{searchApiKey}}
```

```
{
```

```
    "name": "doc-extraction-multimodal-embedding-index",
```

```
    "fields": [
```

```
        {
```

```
            "name": "content_id",
```

```
            "type": "Edm.String",
```

```
            "retrievable": true,
```

```
            "key": true,
```

```
            "analyzer": "keyword"
```

```
        },
```

```
        {
```

```
            "name": "text_document_id",
```

```
            "type": "Edm.String",
```

```
            "searchable": false,
```

```
            "filterable": true,
```

```
            "retrievable": true,
```

```
            "stored": true,
```

```
            "sortable": false,
```

```
            "facetable": false
```

```
        },
```

```
        {
```

```
            "name": "document_title",
```

```
            "type": "Edm.String",
```

```
            "searchable": true
```

```
        },
```

```
        {
```

```
            "name": "image_document_id",
```

```
            "type": "Edm.String",
```

```
            "filterable": true,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "content_text",
```

```
            "type": "Edm.String",
```

```
            "searchable": true,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "content_embedding",
```

```
            "type": "Collection(Edm.Single)",
```

```
            "dimensions": 1024,
```

```
            "searchable": true,
```

```
            "retrievable": true,
```

```
            "vectorSearchProfile": "hnsw"
```

```
        },
```

```
        {
```

```
            "name": "content_path",
```

```
            "type": "Edm.String",
```

```
            "searchable": false,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "offset",
```

```
        "type": "Edm.String",
        "searchable": false,
        "retrievable": true
    },
    {
        "name": "location_metadata",
        "type": "Edm.ComplexType",
        "fields": [
            {
                "name": "page_number",
                "type": "Edm.Int32",
                "searchable": false,
                "retrievable": true
            },
            {
                "name": "bounding_polygons",
                "type": "Edm.String",
                "searchable": false,
                "retrievable": true,
                "filterable": false,
                "sortable": false,
                "facetable": false
            }
        ]
    }
],
"vectorSearch": {
    "profiles": [
        {
            "name": "hnsw",
            "algorithm": "defaulthnsw",
            "vectorizer": "demo-vectorizer"
        }
    ],
    "algorithms": [
        {
            "name": "defaulthnsw",
            "kind": "hnsw",
            "hnswParameters": {
                "m": 4,
                "efConstruction": 400,
                "metric": "cosine"
            }
        }
    ],
    "vectorizers": [
        {
            "name": "demo-vectorizer",
            "kind": "aiServicesVision",
            "aiServicesVisionParameters": {
                "resourceUri": "{{cognitiveServicesUrl}}",
                "authIdentity": null,
                "modelVersion": "{{modelVersion}}"
            }
        }
    ]
}
```

```

        ],
    },
    "semantic": {
        "defaultConfiguration": "semanticconfig",
        "configurations": [
            {
                "name": "semanticconfig",
                "prioritizedFields": {
                    "titleField": {
                        "fieldName": "document_title"
                    },
                    "prioritizedContentFields": [
                    ],
                    "prioritizedKeywordsFields": []
                }
            }
        ]
    }
}

```

Key points:

- Text and image embeddings are stored in the `content_embedding` field and must be configured with appropriate dimensions, such as 1024, and a vector search profile.
- `location_metadata` captures bounding polygon and page number metadata for each normalized image, enabling precise spatial search or UI overlays. `location_metadata` only exists for images in this scenario. If you'd like to capture locational metadata for text as well, consider using [Document Layout skill](#). An in-depth tutorial is linked at the bottom of the page.
- For more information on vector search, see [Vectors in Azure AI Search](#).
- For more information on semantic ranking, see [Semantic ranking in Azure AI Search](#)

Create a skillset

[Create Skillset \(REST\)](#) creates a skillset on your search service. A skillset defines the operations that chunk and embed content prior to indexing. This skillset uses the built-in Document Extraction skill to extract text and images. It uses Text Split skill to chunk large text. It uses Azure Vision multimodal embeddings skill to vectorize image and text content.

HTTP

```

### Create a skillset
POST {{searchUrl}}/skillsets?api-version=2025-11-01-preview   HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

```

```
{
  "name": "doc-extraction-multimodal-embedding-skillset",
  "description": "A test skillset",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Util.DocumentExtractionSkill",
      "name": "document-extraction-skill",
      "description": "Document extraction skill to extract text and images from documents",
      "parsingMode": "default",
      "dataToExtract": "contentAndMetadata",
      "configuration": {
        "imageAction": "generateNormalizedImages",
        "normalizedImageMaxWidth": 2000,
        "normalizedImageMaxHeight": 2000
      },
      "context": "/document",
      "inputs": [
        {
          "name": "file_data",
          "source": "/document/file_data"
        }
      ],
      "outputs": [
        {
          "name": "content",
          "targetName": "extracted_content"
        },
        {
          "name": "normalized_images",
          "targetName": "normalized_images"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
      "name": "split-skill",
      "description": "Split skill to chunk documents",
      "context": "/document",
      "defaultLanguageCode": "en",
      "textSplitMode": "pages",
      "maximumPageLength": 2000,
      "pageOverlapLength": 200,
      "unit": "characters",
      "inputs": [
        {
          "name": "text",
          "source": "/document/extracted_content",
          "inputs": []
        }
      ],
      "outputs": [
        {
          "name": "textItems",
        }
      ]
    }
  ]
}
```

```
        "targetName": "pages"
    }
]
},
{
"@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
"name": "text-embedding-skill",
"description": "Vision Vectorization skill for text",
"context": "/document/pages/*",
"modelVersion": "{{modelVersion}}",
"inputs": [
{
    "name": "text",
    "source": "/document/pages/*"
}
],
"outputs": [
{
    "name": "vector",
    "targetName": "text_vector"
}
]
},
{
"@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
"name": "image-embedding-skill",
"description": "Vision Vectorization skill for images",
"context": "/document/normalized_images/*",
"modelVersion": "{{modelVersion}}",
"inputs": [
{
    "name": "image",
    "source": "/document/normalized_images/*"
}
],
"outputs": [
{
    "name": "vector",
    "targetName": "image_vector"
}
]
},
{
"@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
"name": "shaper-skill",
"description": "Shaper skill to reshape the data to fit the index schema",
"context": "/document/normalized_images/*",
"inputs": [
{
    "name": "normalized_images",
    "source": "/document/normalized_images/*",
    "inputs": []
},
{
    "name": "imagePath",
    "source": "/document/normalized_images/*"
}
]
```

```
        "source":  
"='{{imageProjectionContainer}}/'+$(/document/normalized_images/*/imagePath)",  
        "inputs": []  
    },  
    {  
        "name": "dataUri",  
        "source":  
"='data:image/jpeg;base64,'+$(/document/normalized_images/*/data)",  
        "inputs": []  
    },  
    {  
        "name": "location_metadata",  
        "sourceContext": "/document/normalized_images/*",  
        "inputs": [  
            {  
                "name": "page_number",  
                "source": "/document/normalized_images/*/pageNumber"  
            },  
            {  
                "name": "bounding_polygons",  
                "source": "/document/normalized_images/*/boundingPolygon"  
            }  
        ]  
    },  
    "outputs": [  
        {  
            "name": "output",  
            "targetName": "new_normalized_images"  
        }  
    ]  
},  
],  
"cognitiveServices": {  
    "@odata.type": "#Microsoft.Azure.Search.AIServicesByIdentity",  
    "subdomainUrl": "{{cognitiveServicesUrl}}",  
    "identity": null  
},  
"indexProjections": {  
    "selectors": [  
        {  
            "targetIndexName": "doc-extraction-multimodal-embedding-index",  
            "parentKeyFieldName": "text_document_id",  
            "sourceContext": "/document/pages/*",  
            "mappings": [  
                {  
                    "name": "content_embedding",  
                    "source": "/document/pages/*/text_vector"  
                },  
                {  
                    "name": "content_text",  
                    "source": "/document/pages/*"  
                },  
                {  
                    "name": "document_title",  
                    "source": "/document/pages/*"  
                }  
            ]  
        }  
    ]  
}
```

```

        "source": "/document/document_title"
    }
]
},
{
    "targetIndexName": "doc-extraction-multimodal-embedding-index",
    "parentKeyFieldName": "image_document_id",
    "sourceContext": "/document/normalized_images/*",
    "mappings": [
        {
            "name": "content_embedding",
            "source": "/document/normalized_images/*/image_vector"
        },
        {
            "name": "content_path",
            "source":
"/document/normalized_images/*/new_normalized_images/imagePath"
        },
        {
            "name": "location_metadata",
            "source":
"/document/normalized_images/*/new_normalized_images/location_metadata"
        },
        {
            "name": "document_title",
            "source": "/document/document_title"
        }
    ]
},
],
"parameters": {
    "projectionMode": "skipIndexingParentDocuments"
}
},
"knowledgeStore": {
    "storageConnectionString": "{{storageConnection}}",
    "identity": null,
    "projections": [
        {
            "files": [
                {
                    "storageContainer": "{{imageProjectionContainer}}",
                    "source": "/document/normalized_images/*"
                }
            ]
        }
    ]
}
}
}

```

This skillset extracts text and images, vectorizes both, and shapes the image metadata for projection into the index.

Key points:

- The `content_text` field is populated with text extracted using the Document Extraction Skill and chunked using the Split Skill
- `content_path` contains the relative path to the image file within the designated image projection container. This field is generated only for images extracted from PDFs when `imageAction` is set to `generateNormalizedImages`, and can be mapped from the enriched document from the source field `/document/normalized_images/*/imagePath`.
- The Azure Vision multimodal embeddings skill enables embedding of both textual and visual data using the same skill type, differentiated by input (text vs image). For more information, see [Azure Vision multimodal embeddings skill](#).

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads data, runs a skillset, and indexes the enriched data.

HTTP

```
### Create and run an indexer
POST {{searchUrl}}/indexers?api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-extraction-multimodal-embedding-indexer",
  "dataSourceName": "doc-extraction-multimodal-embedding-ds",
  "targetIndexName": "doc-extraction-multimodal-embedding-index",
  "skillsetName": "doc-extraction-multimodal-embedding-skillset",
  "parameters": {
    "maxFailedItems": -1,
    "maxFailedItemsPerBatch": 0,
    "batchSize": 1,
    "configuration": {
      "allowSkillsetToReadFileData": true
    }
  },
  "fieldMappings": [
    {
      "sourceFieldName": "metadata_storage_name",
      "targetFieldName": "document_title"
    }
  ],
  "outputFieldMappings": []
}
```

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```
### Query the index
POST {{searchUrl}}/indexes/doc-extraction-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

{
  "search": "*",
  "count": true
}
```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 712ca003-9493-40f8-a15e-cf719734a805
elapsed-time: 198
Date: Wed, 30 Apr 2025 23:20:53 GMT
Connection: close

{
  "@odata.count": 100,
  "@search.nextPageParameters": {
    "search": "*",
    "count": true,
    "skip": 50
  },
  "value": [
  ],
  "@odata.nextLink": "https://<YOUR-SEARCH-SERVICE-
NAME>.search.windows.net/indexes/doc-extraction-multimodal-embedding-
index/docs/search?api-version=2025-11-01-preview "
}
```

100 documents are returned in the response.

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case-sensitive. For more information and examples, see [Examples of simple search queries](#).

⚠ Note

The `$filter` parameter only works on fields that were marked filterable during index creation.

HTTP

```
### Query for only images
POST {{searchUrl}}/indexes/doc-extraction-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{



  "search": "*",
  "count": true,
  "filter": "image_document_id ne null"
}
```

HTTP

```
### Query for text or images with content related to energy, returning the id,
parent document, and text (only populated for text chunks), and the content path
where the image is saved in the knowledge store (only populated for images)
POST {{searchUrl}}/indexes/doc-extraction-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}



{



  "search": "energy",
  "count": true,
  "select": "content_id, document_title, content_text, content_path"
}
```

Reset and rerun

Indexers can be reset to clear the high-water mark, which allows a full rerun. The following POST requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{searchUrl}}/indexers/doc-extraction-multimodal-embedding-indexer/reset?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

HTTP

```
### Run the indexer
POST {{searchUrl}}/indexers/doc-extraction-multimodal-embedding-indexer/run?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

HTTP

```
### Check indexer status
GET {{searchUrl}}/indexers/doc-extraction-multimodal-embedding-indexer/status?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

See also

Now that you're familiar with a sample implementation of a multimodal indexing scenario, check out:

- [Azure Vision multimodal embeddings skill](#)
- [Vectors in Azure AI Search](#)
- [Semantic ranking in Azure AI Search](#)
- [Tutorial: Verbalize images from a structured document layout](#)

Tutorial: Vectorize from a structured document layout

Azure AI Search can extract and index both text and images from PDF documents stored in Azure Blob Storage. This tutorial shows you how to build a multimodal indexing pipeline that *chunks data based on document structure* and *uses multimodal embeddings* to vectorize text and images from the same document. Cropped images are stored in a knowledge store, and both text and visual content are vectorized and ingested in a searchable index. Chunking is based on the [Azure Document Intelligence in Foundry Tools layout model](#) that recognizes document structure.

In this tutorial, you use:

- A 36-page PDF document that combines rich visual content, such as charts, infographics, and scanned pages, with traditional text.
- An indexer and skillset to create an indexing pipeline that includes AI enrichment through skills.
- The [Document Layout skill](#) for extracting text and normalized images with its `locationMetadata` from various documents, such as page numbers or bounding regions.
- The [Azure Vision multimodal embeddings skill](#) to vectorize text and images.
- A search index configured to store extracted text and image content. Some content is vectorized for vector-based similarity search.

Prerequisites

- [Microsoft Foundry resource](#). This resource provides access to both the Azure Vision multimodal embedding model and the Azure Document Intelligence Layout model used by the skills in this tutorial. You must use a Foundry resource for skillset access to these resources.
- [Azure AI Search](#). [Configure your search service](#) for role-based access control and a managed identity. Your service must be on the Basic tier or higher. This tutorial isn't supported on the Free tier.
- [Azure Storage](#), used for storing sample data and for creating a [knowledge store](#).
- [Visual Studio Code](#) with a [REST client](#).

Limitations

- The [Document Layout skill](#) has limited regional availability. When you create the Foundry resource, choose a region that provides multimodal embeddings. For a list of supported regions, see [Document Layout skill supported regions](#).
- The [Azure Vision multimodal embeddings skill](#) also has limited regional availability. For an updated list of regions that provide multimodal embeddings, see the [Azure Vision documentation](#).

Prepare data

The following instructions apply to Azure Storage which provides the sample data and also hosts the knowledge store. A search service identity needs read access to Azure Storage to retrieve the sample data, and it needs write access to create the knowledge store. The search service creates the container for cropped images during skillset processing, using the name you provide in an environment variable.

1. Download the following sample PDF: [sustainable-ai-pdf](#)
2. In Azure Storage, create a new container named **sustainable-ai-pdf**.
3. [Upload the sample data file](#).
4. [Create role assignments and specify a managed identity in a connection string](#):
 - a. Assign **Storage Blob Data Reader** for data retrieval by the indexer. Assign **Storage Blob Data Contributor** and **Storage Table Data Contributor** to create and load the knowledge store. You can use either a system-assigned managed identity or a user-assigned managed identity for your search service role assignment.
 - b. For connections made using a system-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-0000-  
    00000000/resourceGroups/MY-DEMO-RESOURCE-  
    GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/;"  
}
```

c. For connections made using a user-assigned managed identity, get a connection string that contains a ResourceId, with no account key or password. The ResourceId must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. Provide an identity using the syntax shown in the following example. Set userAssignedIdentity to the user-assigned managed identity. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-0000-  
00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/ ;"  
},  
"identity" : {  
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",  
    "userAssignedIdentity" : "/subscriptions/00000000-0000-0000-0000-  
00000000/resourcegroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.ManagedIdentity/userAssignedIdentities/MY-DEMO-  
USER-MANAGED-IDENTITY"  
}
```

Prepare models

This tutorial assumes you have an existing Foundry resource through which the skill calls the Azure Vision multimodal 4.0 embedding model. The search service connects to the model during skillset processing using its managed identity. This section gives you guidance and links for assigning roles for authorized access.

The same role assignment is also used for accessing the Azure Document Intelligence layout model via a Foundry resource.

1. Sign in to the Azure portal (not the Foundry portal) and find the Foundry resource. Make sure it's in a region that provides the [multimodal 4.0 API](#) and the [Azure Document Intelligence layout model](#).
2. Select **Access control (IAM)**.
3. Select **Add** and then **Add role assignment**.
4. Search for **Cognitive Services User** and then select it.
5. Choose **Managed identity** and then assign your [search service managed identity](#).

Set up your REST file

For this tutorial, your local REST client connection to Azure AI Search requires an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Connect to a search service](#).

For authenticated connections that occur during indexer and skillset processing, the search service uses the role assignments you previously defined.

1. Start Visual Studio Code and create a new file.
2. Provide values for variables used in the request. For `@storageConnection`, make sure your connection string doesn't have a trailing semicolon or quotation marks. For `@imageProjectionContainer`, provide a container name that's unique in blob storage. Azure AI Search creates this container for you during skills processing.

HTTP

```
@searchUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE  
@searchApiKey = PUT-YOUR-ADMIN-API-KEY-HERE  
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE  
@cognitiveServicesUrl = PUT-YOUR-AZURE-AI-FOUNDRY-ENDPOINT-HERE  
@modelVersion = 2023-04-15  
@imageProjectionContainer=sustainable-ai-pdf-images
```

3. Save the file using a `.rest` or `.http` file extension. For help with the REST client, see [Quickstart: Full-text search using REST](#).

To get the Azure AI Search endpoint and API key:

1. Sign in to the [Azure portal](#), navigate to the search service **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

Create a data source

[Create Data Source \(REST\)](#) creates a data source connection that specifies what data to index.

HTTP

```
### Create a data source using system-assigned managed identities
POST {{searchUrl}}/datasources?api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-intelligence-multimodal-embedding-ds",
  "description": "A data source to store multimodal documents",
  "type": "azureblob",
  "subtype": null,
  "credentials": {
    "connectionString": "{{storageConnection}}"
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Location: https://<YOUR-SEARCH-SERVICE-NAME>.search.windows-
int.net:443/datasources('doc-extraction-multimodal-embedding-ds')?api-version=2025-
11-01-preview -Preview
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 4eb8bcc3-27b5-44af-834e-295ed078e8ed
elapsed-time: 346
Date: Sat, 26 Apr 2025 21:25:24 GMT
Connection: close

{
  "name": "doc-extraction-multimodal-embedding-ds",
  "description": null,
  "type": "azureblob",
  "subtype": null,
  "indexerPermissionOptions": [],
  "credentials": {
    "connectionString": null
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the parameters and their attributes.

For nested JSON, the index fields must be identical to the source fields. Currently, Azure AI Search doesn't support field mappings to nested JSON, so field names and data types must match completely. The following index aligns to the JSON elements in the raw content.

HTTP

```
### Create an index
POST {{searchUrl}}/indexes?api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
```

```
api-key: {{searchApiKey}}
```

```
{
```

```
    "name": "doc-intelligence-multimodal-embedding-index",
```

```
    "fields": [
```

```
        {
```

```
            "name": "content_id",
```

```
            "type": "Edm.String",
```

```
            "retrievable": true,
```

```
            "key": true,
```

```
            "analyzer": "keyword"
```

```
        },
```

```
        {
```

```
            "name": "text_document_id",
```

```
            "type": "Edm.String",
```

```
            "searchable": false,
```

```
            "filterable": true,
```

```
            "retrievable": true,
```

```
            "stored": true,
```

```
            "sortable": false,
```

```
            "facetable": false
```

```
        },
```

```
        {
```

```
            "name": "document_title",
```

```
            "type": "Edm.String",
```

```
            "searchable": true
```

```
        },
```

```
        {
```

```
            "name": "image_document_id",
```

```
            "type": "Edm.String",
```

```
            "filterable": true,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "content_text",
```

```
            "type": "Edm.String",
```

```
            "searchable": true,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "content_embedding",
```

```
            "type": "Collection(Edm.Single)",
```

```
            "dimensions": 1024,
```

```
            "searchable": true,
```

```
            "retrievable": true,
```

```
            "vectorSearchProfile": "hnsw"
```

```
        },
```

```
        {
```

```
            "name": "content_path",
```

```
            "type": "Edm.String",
```

```
            "searchable": false,
```

```
            "retrievable": true
```

```
        },
```

```
        {
```

```
            "name": "offset",
```

```
        "type": "Edm.String",
        "searchable": false,
        "retrievable": true
    },
    {
        "name": "location_metadata",
        "type": "Edm.ComplexType",
        "fields": [
            {
                "name": "page_number",
                "type": "Edm.Int32",
                "searchable": false,
                "retrievable": true
            },
            {
                "name": "bounding_polygons",
                "type": "Edm.String",
                "searchable": false,
                "retrievable": true,
                "filterable": false,
                "sortable": false,
                "facetable": false
            }
        ]
    }
],
"vectorSearch": {
    "profiles": [
        {
            "name": "hnsw",
            "algorithm": "defaulthnsw",
            "vectorizer": "demo-vectorizer"
        }
    ],
    "algorithms": [
        {
            "name": "defaulthnsw",
            "kind": "hnsw",
            "hnswParameters": {
                "m": 4,
                "efConstruction": 400,
                "metric": "cosine"
            }
        }
    ],
    "vectorizers": [
        {
            "name": "demo-vectorizer",
            "kind": "aiServicesVision",
            "aiServicesVisionParameters": {
                "resourceUri": "{{cognitiveServicesUrl}}",
                "authIdentity": null,
                "modelVersion": "{{modelVersion}}"
            }
        }
    ]
}
```

```

        ]
    },
    "semantic": {
        "defaultConfiguration": "semanticconfig",
        "configurations": [
            {
                "name": "semanticconfig",
                "prioritizedFields": {
                    "titleField": {
                        "fieldName": "document_title"
                    },
                    "prioritizedContentFields": [
                    ],
                    "prioritizedKeywordsFields": []
                }
            }
        ]
    }
}

```

Key points:

- Text and image embeddings are stored in the `content_embedding` field and must be configured with appropriate dimensions, such as 1024, and a vector search profile.
- `location_metadata` captures bounding polygon and page number metadata for each text chunk and normalized image, enabling precise spatial search or UI overlays.
- For more information on vector search, see [Vectors in Azure AI Search](#).
- For more information on semantic ranking, see [Semantic ranking in Azure AI Search](#).

Create a skillset

[Create Skillset \(REST\)](#) creates a skillset on your search service. A skillset defines the operations that chunk and embed content prior to indexing. This skillset uses the Document Layout skill to extract text and images, preserving location metadata which is useful for citations in RAG applications. It uses Azure Vision multimodal embeddings skill to vectorize image and text content.

HTTP

```

### Create a skillset
POST {{searchUrl}}/skillsets?api-version=2025-11-01-preview   HTTP/1.1
  Content-Type: application/json
  api-key: {{searchApiKey}}


{
    "name": "doc-intelligence-multimodal-embedding-skillset",

```

```
"description": "A sample skillset for multimodal using multimodal embedding",
"skills": [
  {
    "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",
    "name": "document-layout-skill",
    "description": "Azure Document Intelligence skill for document cracking",
    "context": "/document",
    "outputMode": "oneToMany",
    "outputFormat": "text",
    "extractionOptions": ["images", "locationMetadata"],
    "chunkingProperties": {
      "unit": "characters",
      "maxLength": 2000,
      "overlapLength": 200
    },
    "inputs": [
      {
        "name": "file_data",
        "source": "/document/file_data"
      }
    ],
    "outputs": [
      {
        "name": "text_sections",
        "targetName": "text_sections"
      },
      {
        "name": "normalized_images",
        "targetName": "normalized_images"
      }
    ]
  },
  {
    "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
    "name": "text-embedding-skill",
    "description": "Vision Vectorization skill for text",
    "context": "/document/text_sections/*",
    "modelVersion": "2023-04-15",
    "inputs": [
      {
        "name": "text",
        "source": "/document/text_sections/*/content"
      }
    ],
    "outputs": [
      {
        "name": "vector",
        "targetName": "text_vector"
      }
    ]
  },
  {
    "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
    "name": "image-embedding-skill",
    "description": "Vision Vectorization skill for images",
```

```
"context": "/document/normalized_images/*",
"modelVersion": "2023-04-15",
"inputs": [
  {
    "name": "image",
    "source": "/document/normalized_images/*"
  }
],
"outputs": [
  {
    "name": "vector",
    "targetName": "image_vector"
  }
]
},
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "shaper-skill",
  "context": "/document/normalized_images/*",
  "inputs": [
    {
      "name": "normalized_images",
      "source": "/document/normalized_images/*",
      "inputs": []
    },
    {
      "name": "imagePath",
      "source": "'=my_container_name/'+$(/document/normalized_images/*/imagePath)",
      "inputs": []
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "new_normalized_images"
    }
  ]
},
"indexProjections": {
  "selectors": [
    {
      "targetIndexName": "doc-intelligence-multimodal-embedding-index",
      "parentKeyFieldName": "text_document_id",
      "sourceContext": "/document/text_sections/*",
      "mappings": [
        {
          "name": "content_embedding",
          "source": "/document/text_sections/*/text_vector"
        },
        {
          "name": "content_text",
          "source": "/document/text_sections/*/content"
        },
        {
          "name": "content_file",
          "source": "/document/text_sections/*/file"
        }
      ]
    }
  ]
}
```

```

        {
            "name": "location_metadata",
            "source": "/document/text_sections/*/locationMetadata"
        },
        {
            "name": "document_title",
            "source": "/document/document_title"
        }
    ]
},
{
    "targetIndexName": "{{index}}",
    "parentKeyFieldName": "image_document_id",
    "sourceContext": "/document/normalized_images/*",
    "mappings": [
        {
            "name": "content_embedding",
            "source": "/document/normalized_images/*/image_vector"
        },
        {
            "name": "content_path",
            "source":
"/document/normalized_images/*/new_normalized_images/imagePath"
        },
        {
            "name": "document_title",
            "source": "/document/document_title"
        },
        {
            "name": "location_metadata",
            "source": "/document/normalized_images/*/locationMetadata"
        }
    ]
},
"parameters": {
    "projectionMode": "skipIndexingParentDocuments"
}
},
"cognitiveServices": {
    "@odata.type": "#Microsoft.Azure.Search.AIServicesByIdentity",
    "subdomainUrl": "{{cognitiveServicesUrl}}",
    "identity": null
},
"knowledgeStore": {
    "storageConnectionString": "",
    "identity": null,
    "projections": [
        {
            "files": [
                {
                    "storageContainer": "{{imageProjectionContainer}}",
                    "source": "/document/normalized_images/*"
                }
            ]
        }
    ]
}

```

```
        }
    ]
}
}
```

This skillset extracts text and images, vectorizes both, and shapes the image metadata for projection into the index.

Key points:

- The `content_text` field is populated with text extracted and chunked using the Document Layout Skill
- `content_path` contains the relative path to the image file within the designated image projection container. This field is generated only for images extracted from documents when `extractOption` is set to `["images", "locationMetadata"]` or `["images"]`, and can be mapped from the enriched document from the source field
`/document/normalized_images/*/imagePath.`
- The Azure Vision multimodal embeddings skill enables embedding of both textual and visual data using the same skill type, differentiated by input (text vs image). For more information, see [Azure Vision multimodal embeddings skill](#).

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads data, runs a skillset, and indexes the enriched data.

HTTP

```
### Create and run an indexer
POST {{searchUrl}}/indexers?api-version=2025-11-01-preview   HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "dataSourceName": "doc-intelligence-multimodal-embedding-ds",
  "targetIndexName": "doc-intelligence-multimodal-embedding-index",
  "skillsetName": "doc-intelligence-multimodal-embedding-skillset",
  "parameters": {
    "maxFailedItems": -1,
    "maxFailedItemsPerBatch": 0,
    "batchSize": 1,
    "configuration": {
      "allowSkillsetToReadFileData": true
    }
  },
}
```

```
"fieldMappings": [
  {
    "sourceFieldName": "metadata_storage_name",
    "targetFieldName": "document_title"
  }
],
"outputFieldMappings": []
}
```

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```
### Query the index
POST {{searchUrl}}/indexes/doc-intelligence-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "search": "*",
  "count": true
}
```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
{
  "@odata.count": 100,
  "@search.nextPageParameters": {
    "search": "*",
    "count": true,
    "skip": 50
  },
  "value": [
  ],
  "@odata.nextLink": "https://<YOUR-SEARCH-SERVICE-
NAME>.search.windows.net/indexes/doc-intelligence-multimodal-embedding-
index/docs/search?api-version=2025-11-01-preview "
}
```

100 documents are returned in the response.

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case-sensitive. For more information and examples, see [Examples of simple search queries](#).

⚠ Note

The `$filter` parameter only works on fields that were marked filterable during index creation.

HTTP

```
### Query for only images
POST {{searchUrl}}/indexes/doc-intelligence-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

{
  "search": "*",
  "count": true,
  "filter": "image_document_id ne null"
}
```

HTTP

```
### Query for text or images with content related to energy, returning the id,
parent document, and text (only populated for text chunks), and the content path
where the image is saved in the knowledge store (only populated for images)
POST {{searchUrl}}/indexes/doc-intelligence-multimodal-embedding-index/docs/search?
api-version=2025-11-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

{
  "search": "energy",
  "count": true,
  "select": "content_id, document_title, content_text, content_path"
}
```

Reset and rerun

Indexers can be reset to clear execution history, which allows a full rerun. The following POST requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{searchUrl}}/indexers/doc-intelligence-multimodal-embedding-indexer/reset?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

HTTP

```
### Run the indexer
POST {{searchUrl}}/indexers/doc-intelligence-multimodal-embedding-indexer/run?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

HTTP

```
### Check indexer status
GET {{searchUrl}}/indexers/doc-intelligence-multimodal-embedding-indexer/status?api-version=2025-11-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

See also

Now that you're familiar with a sample implementation of a multimodal indexing scenario, check out:

- [Azure Vision multimodal embeddings skill](#)
- [Document Layout skill](#)
- [Vectors in Azure AI Search](#)
- [Semantic ranking in Azure AI Search](#)
- [Tutorial: Vectorize images and text](#)

Tutorial: Verbalize images using generative AI

08/27/2025

Azure AI Search can extract and index both text and images from PDF documents stored in Azure Blob Storage. This tutorial shows you how to build a multimodal indexing pipeline that *chunks data using the built-in Text Split skill* and uses *image verbalization* to describe images. Cropped images are stored in a knowledge store, and visual content is described in natural language and ingested alongside text in a searchable index.

To get image verbalizations, each extracted image is passed to the [GenAI Prompt skill \(preview\)](#) that calls a chat completion model to generate a concise textual description. These descriptions, along with the original document text, are then embedded into vector representations using Azure OpenAI's text-embedding-3-large model. The result is a single index containing searchable content from both modalities: text and verbalized images.

In this tutorial, you use:

- A 36-page PDF document that combines rich visual content, such as charts, infographics, and scanned pages, with traditional text.
- An indexer and skillset to create an indexing pipeline that includes AI enrichment through skills.
- The [Document Extraction skill](#) for extracting normalized images and text. The [Text Split skill](#) chunks the data.
- The [GenAI Prompt skill \(preview\)](#) that calls a chat completion model to create descriptions of visual content.
- A search index configured to store text and image verbalizations. Some content is vectorized for vector-based similarity search.

This tutorial demonstrates a lower-cost approach for indexing multimodal content using the Document Extraction skill and image captioning. It enables extraction and search over both text and images from documents in Azure Blob Storage. However, it doesn't include locational metadata for text, such as page numbers or bounding regions. For a more comprehensive solution that includes structured text layout and spatial metadata, see [Tutorial: Verbalize images from a structured document layout](#).

Note

Image extraction by the Document Extraction skill isn't free. Setting `imageAction` to `generateNormalizedImages` in the skillset triggers image extraction, which is an extra charge. For billing information, see [Azure AI Search pricing](#).

Prerequisites

- [Azure AI Search](#). Configure your search service for role-based access control and a managed identity. Your service must be on the Basic tier or higher. This tutorial isn't supported on the Free tier.
- [Azure Storage](#), used for storing sample data and for creating a [knowledge store](#).
- [Azure OpenAI](#) with a deployment of
 - A chat completion model hosted in Azure AI Foundry or another source. The model is used to verbalize image content. You provide the URI to the hosted model in the GenAI Prompt skill definition. You can use [any chat completion model](#).
 - A text embedding model deployed in Azure AI Foundry. The model is used to vectorize text content pull from source documents and the image descriptions generated by the chat completion model. For integrated vectorization, the embedding model must be located in Azure AI Foundry, and it must be either text-embedding-ada-002, text-embedding-3-large, or text-embedding-3-small. If you want to use an external embedding model, use a custom skill instead of the Azure OpenAI embedding skill.
- [Visual Studio Code](#) with a [REST client](#).

Prepare data

The following instructions apply to Azure Storage which provides the sample data and also hosts the knowledge store. A search service identity needs read access to Azure Storage to retrieve the sample data, and it needs write access to create the knowledge store. The search service creates the container for cropped images during skillset processing, using the name you provide in an environment variable.

1. Download the following sample PDF: [sustainable-ai-pdf](#)
2. In Azure Storage, create a new container named **sustainable-ai-pdf**.
3. [Upload the sample data file](#).
4. [Create role assignments and specify a managed identity in a connection string](#):

- a. Assign **Storage Blob Data Reader** for data retrieval by the indexer. Assign **Storage Blob Data Contributor** and **Storage Table Data Contributor** to create and load the knowledge store. You can use either a system-assigned managed identity or a user-assigned managed identity for your search service role assignment.
- b. For connections made using a system-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
0000-00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-  
ACCOUNT/;"  
}
```

- c. For connections made using a user-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. Provide an identity using the syntax shown in the following example. Set userAssignedIdentity to the user-assigned managed identity. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
0000-00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-  
ACCOUNT/;"  
,  
    "identity" : {  
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",  
        "userAssignedIdentity" : "/subscriptions/00000000-0000-0000-  
00000000/resourcegroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.ManagedIdentity/userAssignedIdentities/MY-DEMO-  
USER-MANAGED-IDENTITY"  
    }  
}
```

Prepare models

This tutorial assumes you have an existing Azure OpenAI resource through which the skills call the text embedding model and chat completion models. The search service connects to the models during skillset processing and during query execution using its managed identity. This section gives you guidance and links for assigning roles for authorized access.

1. Sign in to the Azure portal (not the Foundry portal) and find the Azure OpenAI resource.
2. Select **Access control (IAM)**.
3. Select **Add** and then **Add role assignment**.
4. Search for **Cognitive Services OpenAI User** and then select it.
5. Choose **Managed identity** and then assign your [search service managed identity](#).

For more information, see [Role-based access control for Azure OpenAI in Azure AI Foundry Models](#).

Set up your REST file

For this tutorial, your local REST client connection to Azure AI Search requires an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Connect to a search service](#).

For authenticated connections that occur during indexer and skillset processing, the search service uses the role assignments you previously defined.

1. Start Visual Studio Code and create a new file.
2. Provide values for variables used in the request. For `@storageConnection`, make sure your connection string doesn't have a trailing semicolon or quotation marks. For `@imageProjectionContainer`, provide a container name that's unique in blob storage. Azure AI Search creates this container for you during skills processing.

HTTP

```
@searchUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE  
@searchApiKey = PUT-YOUR-ADMIN-API-KEY-HERE  
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE  
@openAIResourceUri = PUT-YOUR-OPENAI-URI-HERE  
@openAIKey = PUT-YOUR-OPENAI-KEY-HERE  
@chatCompletionResourceUri = PUT-YOUR-CHAT-COMPLETION-URI-HERE  
@chatCompletionKey = PUT-YOUR-CHAT-COMPLETION-KEY-HERE  
@imageProjectionContainer=sustainable-ai-pdf-images
```

3. Save the file using a `.rest` or `.http` file extension. For help with the REST client, see [Quickstart: Full-text search using REST](#).

To get the Azure AI Search endpoint and API key:

1. Sign in to the [Azure portal](#), navigate to the search service **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

The screenshot shows two stacked Azure portal pages. The top page is the 'new-demo-search-svc | Overview' page, where the 'Overview' tab is selected (marked with a red box and number 1). The URL `https://new-demo-search-svc.search.windows.net` is displayed in the top right. The bottom page is the 'new-demo-search-svc | Keys' page, where the 'Keys' tab is selected (marked with a red box and number 2). It shows the 'API access control' settings (radio buttons for API keys, Role-based access control, or Both) and the 'Manage admin keys' section, which contains fields for Primary admin key and Secondary admin key, each with a placeholder value like '`<your-primary-admin-key-here>`' and a 'Regenerate' button. Both the URL and the 'Manage admin keys' section are highlighted with red boxes.

Create a data source

[Create Data Source \(REST\)](#) creates a data source connection that specifies what data to index.

```
HTTP

### Create a data source
POST {{searchUrl}}/datasources?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-extraction-image-verbalization-ds",
  "description": null,
  "type": "azureblob",
  "subtype": null,
  "credentials":{
    "connectionString": "{{storageConnection}}"
}
```

```
},
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Location: https://<YOUR-SEARCH-SERVICE-NAME>.search.windows-
int.net:443/datasources('doc-extraction-multimodal-embedding-ds')?api-
version=2025-08-01-preview -Preview
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 4eb8bcc3-27b5-44af-834e-295ed078e8ed
elapsed-time: 346
Date: Sat, 26 Apr 2025 21:25:24 GMT
Connection: close

{
  "name": "doc-extraction-multimodal-embedding-ds",
  "description": null,
  "type": "azureblob",
  "subtype": null,
  "indexerPermissionOptions": [],
  "credentials": {
    "connectionString": null
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the parameters and their attributes.

For nested JSON, the index fields must be identical to the source fields. Currently, Azure AI Search doesn't support field mappings to nested JSON, so field names and data types must match completely. The following index aligns to the JSON elements in the raw content.

HTTP

```
### Create an index
POST {{searchUrl}}/indexes?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-extraction-image-verbalization-index",
  "fields": [
    {
      "name": "content_id",
      "type": "Edm.String",
      " retrievable": true,
      "key": true,
      "analyzer": "keyword"
    },
    {
      "name": "text_document_id",
      "type": "Edm.String",
      "searchable": false,
      "filterable": true,
      "retrievable": true,
      "stored": true,
      "sortable": false,
      "facetable": false
    },
    {
      "name": "document_title",
      "type": "Edm.String",
      "searchable": true
    },
    {
      "name": "image_document_id",
      "type": "Edm.String",
      "filterable": true,
      "retrievable": true
    },
    {
      "name": "content_text",
      "type": "Edm.String",
      "searchable": true,
      "retrievable": true
    },
    {
      "name": "content_embedding",
      "type": "Edm.String",
      "searchable": true,
      "retrievable": true
    }
  ]
}
```

```
        "type": "Collection(Edm.Single)",
        "dimensions": 3072,
        "searchable": true,
        "retrievable": true,
        "vectorSearchProfile": "hnsw"
    },
    {
        "name": "content_path",
        "type": "Edm.String",
        "searchable": false,
        "retrievable": true
    },
    {
        "name": "offset",
        "type": "Edm.String",
        "searchable": false,
        "retrievable": true
    },
    {
        "name": "location_metadata",
        "type": "Edm.ComplexType",
        "fields": [
            {
                "name": "page_number",
                "type": "Edm.Int32",
                "searchable": false,
                "retrievable": true
            },
            {
                "name": "bounding_polygons",
                "type": "Edm.String",
                "searchable": false,
                "retrievable": true,
                "filterable": false,
                "sortable": false,
                "facetable": false
            }
        ]
    }
],
"vectorSearch": {
    "profiles": [
        {
            "name": "hnsw",
            "algorithm": "defaulthnsw",
            "vectorizer": "demo-vectorizer"
        }
    ],
    "algorithms": [
        {
            "name": "defaulthnsw",
            "kind": "hnsw",
            "hnswParameters": {
                "m": 4,
                "efConstruction": 400,
                "efSearch": 100
            }
        }
    ]
}
```

```

        "metric": "cosine"
    }
}
],
"vectorizers": [
{
    "name": "demo-vectorizer",
    "kind": "azureOpenAI",
    "azureOpenAIParameters": {
        "resourceUri": "{{openAIResourceUri}}",
        "deploymentId": "text-embedding-3-large",
        "searchApiKey": "{{openAIKey}}",
        "modelName": "text-embedding-3-large"
    }
}
]
},
"semantic": {
    "defaultConfiguration": "semanticconfig",
    "configurations": [
{
    "name": "semanticconfig",
    "prioritizedFields": {
        "titleField": {
            "fieldName": "document_title"
        },
        "prioritizedContentFields": [
        ],
        "prioritizedKeywordsFields": []
    }
}
]
}
}

```

Key points:

- Text and image embeddings are stored in the `content_embedding` field and must be configured with appropriate dimensions (for example, 3072) and a vector search profile.
- `location_metadata` captures bounding polygon and page number metadata for each normalized image, enabling precise spatial search or UI overlays. `location_metadata` only exists for images in this scenario. If you'd like to capture locational metadata for text as well, consider using [Document Layout skill](#). An in-depth tutorial is linked at the bottom of the page.
- For more information on vector search, see [Vectors in Azure AI Search](#).
- For more information on semantic ranking, see [Semantic ranking in Azure AI Search](#)

Create a skillset

[Create Skillset \(REST\)](#) creates a skillset on your search service. A skillset defines the operations that chunk and embed content prior to indexing. This skillset uses the built-in Document Extraction skill to extract text and images. It uses Text Split skill to chunk large text. It uses Azure OpenAI Embedding skill to vectorize text content.

The skillset also performs actions specific to images. It uses the GenAI Prompt skill to generate image descriptions. It also creates a knowledge store that stores intact images so that you can return them in a query.

HTTP

```
### Create a skillset
POST {{searchUrl}}/skillsets?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-extraction-image-verbalization-skillset",
  "description": "A test skillset",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Util.DocumentExtractionSkill",
      "name": "document-extraction-skill",
      "description": "Document extraction skill to extract text and images from documents",
      "parsingMode": "default",
      "dataToExtract": "contentAndMetadata",
      "configuration": {
        "imageAction": "generateNormalizedImages",
        "normalizedImageMaxWidth": 2000,
        "normalizedImageMaxHeight": 2000
      },
      "context": "/document",
      "inputs": [
        {
          "name": "file_data",
          "source": "/document/file_data"
        }
      ],
      "outputs": [
        {
          "name": "content",
          "targetName": "extracted_content"
        },
        {
          "name": "normalized_images",
          "targetName": "normalized_images"
        }
      ]
    }
  ]
}
```

```
},
{
  "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
  "name": "split-skill",
  "description": "Split skill to chunk documents",
  "context": "/document",
  "defaultLanguageCode": "en",
  "textSplitMode": "pages",
  "maximumPageLength": 2000,
  "pageOverlapLength": 200,
  "unit": "characters",
  "inputs": [
    {
      "name": "text",
      "source": "/document/extracted_content",
      "inputs": []
    }
  ],
  "outputs": [
    {
      "name": "textItems",
      "targetName": "pages"
    }
  ]
},
{
  "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
  "name": "text-embedding-skill",
  "description": "Embedding skill for text",
  "context": "/document/pages/*",
  "inputs": [
    {
      "name": "text",
      "source": "/document/pages/*"
    }
  ],
  "outputs": [
    {
      "name": "embedding",
      "targetName": "text_vector"
    }
  ],
  "resourceUri": "{{openAIResourceUri}}",
  "deploymentId": "text-embedding-3-large",
  "searchApiKey": "{{openAIKey}}",
  "dimensions": 3072,
  "modelName": "text-embedding-3-large"
},
{
  "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",
  "name": "genAI-prompt-skill",
  "description": "GenAI Prompt skill for image verbalization",
  "uri": "{{chatCompletionResourceUri}}",
  "timeout": "PT1M",
  "searchApiKey": "{{chatCompletionKey}}"
}
```

```
"context": "/document/normalized_images/*",
"inputs": [
    {
        "name": "systemMessage",
        "source": "'You are tasked with generating concise, accurate descriptions of images, figures, diagrams, or charts in documents. The goal is to capture the key information and meaning conveyed by the image without including extraneous details like style, colors, visual aesthetics, or size.\n\nInstructions:\nContent Focus: Describe the core content and relationships depicted in the image.\nFor diagrams, specify the main elements and how they are connected or interact.\nFor charts, highlight key data points, trends, comparisons, or conclusions.\nFor figures or technical illustrations, identify the components and their significance.\nClarity & Precision: Use concise language to ensure clarity and technical accuracy. Avoid subjective or interpretive statements.\n\nAvoid Visual Descriptors: Exclude details about:\nColors, shading, and visual styles.\nImage size, layout, or decorative elements.\nFonts, borders, and stylistic embellishments.\nContext: If relevant, relate the image to the broader content of the technical document or the topic it supports.\n\nExample Descriptions:\nDiagram: \"A flowchart showing the four stages of a machine learning pipeline: data collection, preprocessing, model training, and evaluation, with arrows indicating the sequential flow of tasks.\"\n\nChart: \"A bar chart comparing the performance of four algorithms on three datasets, showing that Algorithm A consistently outperforms the others on Dataset 1.\"\n\nFigure: \"A labeled diagram illustrating the components of a transformer model, including the encoder, decoder, self-attention mechanism, and feedforward layers.\"''"
    },
    {
        "name": "userMessage",
        "source": "'Please describe this image.'"
    },
    {
        "name": "image",
        "source": "/document/normalized_images/*/data"
    }
],
"outputs": [
    {
        "name": "response",
        "targetName": "verbalizedImage"
    }
]
},
{
    "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
    "name": "verbalized-image-embedding-skill",
    "description": "Embedding skill for verbalized images",
    "context": "/document/normalized_images/*",
    "inputs": [
        {
            "name": "text",
            "source": "/document/normalized_images/*/verbalizedImage",
            "inputs": []
        }
    ],
    "outputs": [
```

```

    },
    "name": "embedding",
    "targetName": "verbalizedImage_vector"
  },
],
"resourceUri": "{{openAIResourceUri}}",
"deploymentId": "text-embedding-3-large",
"searchApiKey": "{{openAIKey}}",
"dimensions": 3072,
"modelName": "text-embedding-3-large"
},
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "shaper-skill",
  "description": "Shaper skill to reshape the data to fit the index schema",
  "context": "/document/normalized_images/*",
  "inputs": [
    {
      "name": "normalized_images",
      "source": "/document/normalized_images/*",
      "inputs": []
    },
    {
      "name": "imagePath",
      "source": ":" + {{imageProjectionContainer}} + $(/document/normalized_images/*/imagePath),
      "inputs": []
    },
    {
      "name": "location_metadata",
      "sourceContext": "/document/normalized_images/*",
      "inputs": [
        {
          "name": "page_number",
          "source": "/document/normalized_images/*/pageNumber"
        },
        {
          "name": "bounding_polygons",
          "source": "/document/normalized_images/*/boundingPolygon"
        }
      ]
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "new_normalized_images"
    }
  ]
},
"indexProjections": {
  "selectors": [
    {
      "targetIndexName": "doc-extraction-image-verbalization-index",
      "projection": {
        "name": "imageProjection"
      }
    }
  ]
}
]

```

```
"parentKeyFieldName": "text_document_id",
"sourceContext": "/document/pages/*",
"mappings": [
  {
    "name": "content_embedding",
    "source": "/document/pages/*/text_vector"
  },
  {
    "name": "content_text",
    "source": "/document/pages/*"
  },
  {
    "name": "document_title",
    "source": "/document/document_title"
  }
]
},
{
  "targetIndexName": "doc-extraction-image-verbalization-index",
  "parentKeyFieldName": "image_document_id",
  "sourceContext": "/document/normalized_images/*",
  "mappings": [
    {
      "name": "content_text",
      "source": "/document/normalized_images/*/verbalizedImage"
    },
    {
      "name": "content_embedding",
      "source": "/document/normalized_images/*/verbalizedImage_vector"
    },
    {
      "name": "content_path",
      "source":
"/document/normalized_images/*/new_normalized_images/imagePath"
    },
    {
      "name": "document_title",
      "source": "/document/document_title"
    },
    {
      "name": "locationMetadata",
      "source":
"/document/normalized_images/*/new_normalized_images/location_metadata"
    }
  ]
},
"parameters": {
  "projectionMode": "skipIndexingParentDocuments"
}
},
"knowledgeStore": {
  "storageConnectionString": "{{storageConnection}}",
  "identity": null,
  "projections": [
```

```

{
  "files": [
    {
      "storageContainer": "{{imageProjectionContainer}}",
      "source": "/document/normalized_images/*"
    }
  ]
}
}

```

This skillset extracts text and images, vectorizes both, and shapes the image metadata for projection into the index.

Key points:

- The `content_text` field is populated in two ways:
 - From document text extracted using the Document Extraction skill and chunked using the Text Split skill
 - From image content using the GenAI Prompt skill, which generates descriptive captions for each normalized image
- The `content_embedding` field contains 3072-dimensional embeddings for both page text and verbalized image descriptions. These are generated using the text-embedding-3-large model from Azure OpenAI.
- `content_path` contains the relative path to the image file within the designated image projection container. This field is generated only for images extracted from PDFs when `imageAction` is set to `generateNormalizedImages`, and can be mapped from the enriched document from the source field `/document/normalized_images/*/imagePath`.

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads data, runs a skillset, and indexes the enriched data.

HTTP

```

### Create and run an indexer
POST {{searchUrl}}/indexers?api-version=2025-08-01-preview   HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

```

```
{  
  "dataSourceName": "doc-extraction-image-verbalization-ds",  
  "targetIndexName": "doc-extraction-image-verbalization-index",  
  "skillsetName": "doc-extraction-image-verbalization-skillset",  
  "parameters": {  
    "maxFailedItems": -1,  
    "maxFailedItemsPerBatch": 0,  
    "batchSize": 1,  
    "configuration": {  
      "allowSkillsetToReadFileData": true  
    }  
  },  
  "fieldMappings": [  
    {  
      "sourceFieldName": "metadata_storage_name",  
      "targetFieldName": "document_title"  
    }  
  ],  
  "outputFieldMappings": []  
}
```

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```
### Query the index  
POST {{searchUrl}}/indexes/doc-extraction-image-verbalization-index/docs/search?  
api-version=2025-08-01-preview HTTP/1.1  
Content-Type: application/json  
api-key: {{searchApiKey}}  
  
{  
  "search": "*",  
  "count": true  
}
```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
HTTP/1.1 200 OK  
Transfer-Encoding: chunked  
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;  
charset=utf-8
```

```
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 712ca003-9493-40f8-a15e-cf719734a805
elapsed-time: 198
Date: Wed, 30 Apr 2025 23:20:53 GMT
Connection: close

{
    "@odata.count": 100,
    "@search.nextPageParameters": {
        "search": "*",
        "count": true,
        "skip": 50
    },
    "value": [
    ],
    "@odata.nextLink": "https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/doc-extraction-image-verbalization-index/docs/search?api-version=2025-08-01-preview "
}
```

100 documents are returned in the response.

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case -sensitive. For more information and examples, see [Examples of simple search queries](#).

 **Note**

The `$filter` parameter only works on fields that were marked filterable during index creation.

Here are some examples of other queries:

HTTP

```
### Query for only images
POST {{searchUrl}}/indexes/doc-extraction-image-verbalization-index/docs/search?
api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
    "search": "*",
    "count": true,
```

```
        "filter": "image_document_id ne null"
    }
```

HTTP

```
### Query for text or images with content related to energy, returning the id, parent document, and text (extracted text for text chunks and verbalized image text for images), and the content path where the image is saved in the knowledge store (only populated for images)
POST {{searchUrl}}/indexes/doc-extraction-image-verbalization-index/docs/search?api-version=2025-08-01-preview HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "search": "energy",
  "count": true,
  "select": "content_id, document_title, content_text, content_path"
}
```

Reset and rerun

Indexers can be reset to clear the high-water mark, which allows a full rerun. The following POST requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{searchUrl}}/indexers/doc-extraction-image-verbalization-indexer/reset?api-version=2025-08-01-preview HTTP/1.1
api-key: {{searchApiKey}}
```

HTTP

```
### Run the indexer
POST {{searchUrl}}/indexers/doc-extraction-image-verbalization-indexer/run?api-version=2025-08-01-preview HTTP/1.1
api-key: {{searchApiKey}}
```

HTTP

```
### Check indexer status
GET {{searchUrl}}/indexers/doc-extraction-image-verbalization-indexer/status?api-version=2025-08-01-preview HTTP/1.1
api-key: {{searchApiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

See also

Now that you're familiar with a sample implementation of a multimodal indexing scenario, check out:

- [GenAI Prompt skill](#)
- [Vectors in Azure AI Search](#)
- [Semantic ranking in Azure AI Search](#)
- [Tutorial: Verbalize images from a structured document layout](#)

Tutorial: Verbalize images from a structured document layout

09/27/2025

Azure AI Search can extract and index both text and images from PDF documents stored in Azure Blob Storage. This tutorial shows you how to build a multimodal indexing pipeline that *chunks data based on document structure* and uses *image verbalization* to describe images. Cropped images are stored in a knowledge store, and visual content is described in natural language and ingested alongside text in a searchable index. Chunking is based on the Azure AI Document Intelligence Layout model that recognizes document structure.

To get image verbalizations, each extracted image is passed to the [GenAI Prompt skill \(preview\)](#) that calls a chat completion model to generate a concise textual description. These descriptions, along with the original document text, are then embedded into vector representations using Azure OpenAI's text-embedding-3-large model. The result is a single index containing searchable content from both modalities: text and verbalized images.

In this tutorial, you use:

- A 36-page PDF document that combines rich visual content, such as charts, infographics, and scanned pages, with traditional text.
- An indexer and skillset to create an indexing pipeline that includes AI enrichment through skills.
- The [Document Layout skill](#) for extracting text and normalized images with its `locationMetadata` from various documents, such as page numbers or bounding regions.
- The [GenAI Prompt skill \(preview\)](#) that calls a chat completion model to create descriptions of visual content.
- A search index configured to store extracted text and image verbalizations. Some content is vectorized for vector-based similarity search.

Prerequisites

- [Azure AI services multi-service account](#). This account provides access to the Document Intelligence Layout model used in this tutorial. You must use an Azure AI multi-service account for skillset access to this resource.
- [Azure AI Search](#). [Configure your search service](#) for role-based access control and a managed identity. Your service must be on the Basic tier or higher. This tutorial isn't

supported on the Free tier.

- [Azure Storage](#), used for storing sample data and for creating a [knowledge store](#).
- [Azure OpenAI](#) with a deployment of
 - A chat completion model hosted in Azure AI Foundry or another source. The model is used to verbalize image content. You provide the URI to the hosted model in the GenAI Prompt skill definition. You can use [any chat completion model](#).
 - A text embedding model deployed in Azure AI Foundry. The model is used to vectorize text content pull from source documents and the image descriptions generated by the chat completion model. For integrated vectorization, the embedding model must be located in Azure AI Foundry, and it must be either text-embedding-ada-002, text-embedding-3-large, or text-embedding-3-small. If you want to use an external embedding model, use a custom skill instead of the Azure OpenAI embedding skill.
- [Visual Studio Code](#) with a [REST client](#).

Limitations

- The [Document Layout skill](#) has limited regional availability. For a list of supported regions, see [Document Layout skill > Supported regions](#).

Prepare data

The following instructions apply to Azure Storage which provides the sample data and also hosts the knowledge store. A search service identity needs read access to Azure Storage to retrieve the sample data, and it needs write access to create the knowledge store. The search service creates the container for cropped images during skillset processing, using the name you provide in an environment variable.

1. Download the following sample PDF: [sustainable-ai-pdf](#)
2. In Azure Storage, create a new container named **sustainable-ai-pdf**.
3. [Upload the sample data file](#).
4. [Create role assignments and specify a managed identity in a connection string](#):
 - a. Assign **Storage Blob Data Reader** for data retrieval by the indexer. Assign **Storage Blob Data Contributor** and **Storage Table Data Contributor** to create and load the knowledge store. You can use either a system-assigned managed identity or a user-assigned managed identity for your search service role assignment.

- b. For connections made using a system-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
0000-00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-  
ACCOUNT/;"  
}
```

- c. For connections made using a user-assigned managed identity, get a connection string that contains a Resourceld, with no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name. Provide an identity using the syntax shown in the following example. Set userAssignedIdentity to the user-assigned managed identity. The connection string is similar to the following example:

JSON

```
"credentials" : {  
    "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-  
0000-00000000/resourceGroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-  
ACCOUNT/;"  
,  
    "identity" : {  
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",  
        "userAssignedIdentity" : "/subscriptions/00000000-0000-0000-  
00000000/resourcegroups/MY-DEMO-RESOURCE-  
GROUP/providers/Microsoft.ManagedIdentity/userAssignedIdentities/MY-DEMO-  
USER-MANAGED-IDENTITY"  
    }  
}
```

Prepare models

This tutorial assumes you have an existing Azure OpenAI resource through which the skills a chat completion model for GenAI Prompt and also a text embedding model for vectorization. The search service connects to the models during skillset processing and during query execution using its managed identity. This section gives you guidance and links for assigning roles for authorized access.

You also need a role assignment for accessing the Document Intelligence Layout model via an Azure AI multi-service account.

Assign roles in Azure AI multi-service

1. Sign in to the Azure portal (not the Foundry portal) and find the Azure AI multi-service account. Make sure it's in a region that provides the [Document Intelligence Layout model](#).
2. Select **Access control (IAM)**.
3. Select **Add** and then **Add role assignment**.
4. Search for **Cognitive Services User** and then select it.
5. Choose **Managed identity** and then assign your [search service managed identity](#).

Assign roles in Azure OpenAI

1. Sign in to the Azure portal (not the Foundry portal) and find the Azure OpenAI resource.
2. Select **Access control (IAM)**.
3. Select **Add** and then **Add role assignment**.
4. Search for **Cognitive Services OpenAI User** and then select it.
5. Choose **Managed identity** and then assign your [search service managed identity](#).

For more information, see [Role-based access control for Azure OpenAI in Azure AI Foundry Models](#).

Set up your REST file

For this tutorial, your local REST client connection to Azure AI Search requires an endpoint and an API key. You can get these values from the Azure portal. For alternative connection methods, see [Connect to a search service](#).

For authenticated connections that occur during indexer and skillset processing, the search service uses the role assignments you previously defined.

1. Start Visual Studio Code and create a new file.
2. Provide values for variables used in the request. For `@storageConnection`, make sure your connection string doesn't have a trailing semicolon or quotation marks. For

@imageProjectionContainer , provide a container name that's unique in blob storage.

Azure AI Search creates this container for you during skills processing.

```
HTTP

@searchUrl = PUT-YOUR-SEARCH-SERVICE-ENDPOINT-HERE
@searchApiKey = PUT-YOUR-ADMIN-API-KEY-HERE
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE
@cognitiveServicesUrl = PUT-YOUR-AZURE-AI-MULTI-SERVICE-ENDPOINT-HERE
@openAIResourceUri = PUT-YOUR-OPENAI-URI-HERE
@openAIKey = PUT-YOUR-OPENAI-KEY-HERE
@chatCompletionResourceUri = PUT-YOUR-CHAT-COMPLETION-URI-HERE
@chatCompletionKey = PUT-YOUR-CHAT-COMPLETION-KEY-HERE
@imageProjectionContainer=sustainable-ai-pdf-images
```

3. Save the file using a `.rest` or `.http` file extension. For help with the REST client, see [Quickstart: Full-text search using REST](#).

To get the Azure AI Search endpoint and API key:

1. Sign in to the [Azure portal](#), navigate to the search service **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

The image contains two screenshots of the Microsoft Azure portal. The top screenshot shows the 'new-demo-search-svc' search service in the 'Overview' tab. A red box highlights the 'Overview' link in the left sidebar, and another red box highlights the 'Url' field which contains the value `https://new-demo-search-svc.search.windows.net`. The bottom screenshot shows the 'new-demo-search-svc | Keys' settings page. A red box highlights the 'Keys' link in the left sidebar, and another red box highlights the 'Primary admin key' input field which contains the placeholder text `<your-primary-admin-key-here>`. Both screenshots show the Azure navigation bar at the top.

Create a data source

Create Data Source (REST) creates a data source connection that specifies what data to index.

HTTP

```
### Create a data source using system-assigned managed identities
POST {{searchUrl}}/datasources?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-intelligence-image-verbalization-ds",
  "description": "A data source to store multi-modality documents",
  "type": "azureblob",
  "subtype": null,
  "credentials": {
    "connectionString": "{{storageConnection}}"
  },
  "container": {
    "name": "sustainable-ai-pdf",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

Send the request. The response should look like:

JSON

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true;
charset=utf-8
Location: https://<YOUR-SEARCH-SERVICE-NAME>.search.windows-
int.net:443/datasources('doc-extraction-multimodal-embedding-ds')?api-
version=2025-08-01-preview -Preview
Server: Microsoft-IIS/10.0
Strict-Transport-Security: max-age=2592000, max-age=15724800; includeSubDomains
Preference-Applied: odata.include-annotations="*"
OData-Version: 4.0
request-id: 4eb8bcc3-27b5-44af-834e-295ed078e8ed
elapsed-time: 346
Date: Sat, 26 Apr 2025 21:25:24 GMT
Connection: close

{
  "name": "doc-extraction-multimodal-embedding-ds",
  "description": null,
  "type": "azureblob",
```

```
"subtype": null,
"indexerPermissionOptions": [],
"credentials": {
    "connectionString": null
},
"container": {
    "name": "sustainable-ai-pdf",
    "query": null
},
"dataChangeDetectionPolicy": null,
"dataDeletionDetectionPolicy": null,
"encryptionKey": null,
"identity": null
}
```

Create an index

[Create Index \(REST\)](#) creates a search index on your search service. An index specifies all the parameters and their attributes.

For nested JSON, the index fields must be identical to the source fields. Currently, Azure AI Search doesn't support field mappings to nested JSON, so field names and data types must match completely. The following index aligns to the JSON elements in the raw content.

HTTP

```
### Create an index
POST {{searchUrl}}/indexes?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
    "name": "doc-intelligence-image-verbalization-index",
    "fields": [
        {
            "name": "content_id",
            "type": "Edm.String",
            "retrievable": true,
            "key": true,
            "analyzer": "keyword"
        },
        {
            "name": "text_document_id",
            "type": "Edm.String",
            "searchable": false,
            "filterable": true,
            "retrievable": true,
            "stored": true,
            "sortable": false,
            "facetable": false
        }
    ]
}
```

```
},
{
    "name": "document_title",
    "type": "Edm.String",
    "searchable": true
},
{
    "name": "image_document_id",
    "type": "Edm.String",
    "filterable": true,
    "retrievable": true
},
{
    "name": "content_text",
    "type": "Edm.String",
    "searchable": true,
    "retrievable": true
},
{
    "name": "content_embedding",
    "type": "Collection(Edm.Single)",
    "dimensions": 3072,
    "searchable": true,
    "retrievable": true,
    "vectorSearchProfile": "hnsw"
},
{
    "name": "content_path",
    "type": "Edm.String",
    "searchable": false,
    "retrievable": true
},
{
    "name": "offset",
    "type": "Edm.String",
    "searchable": false,
    "retrievable": true
},
{
    "name": "location_metadata",
    "type": "Edm.ComplexType",
    "fields": [
        {
            "name": "page_number",
            "type": "Edm.Int32",
            "searchable": false,
            "retrievable": true
        },
        {
            "name": "bounding_polygons",
            "type": "Edm.String",
            "searchable": false,
            "retrievable": true,
            "filterable": false,
            "sortable": false,
            "indexable": true
        }
    ]
}
```

```
        "facetable": false
    }
]
},
],
"vectorSearch": {
    "profiles": [
        {
            "name": "hnsw",
            "algorithm": "defaulthnsw",
            "vectorizer": "demo-vectorizer"
        }
    ],
    "algorithms": [
        {
            "name": "defaulthnsw",
            "kind": "hnsw",
            "hnswParameters": {
                "m": 4,
                "efConstruction": 400,
                "metric": "cosine"
            }
        }
    ],
    "vectorizers": [
        {
            "name": "demo-vectorizer",
            "kind": "azureOpenAI",
            "azureOpenAIParameters": {
                "resourceUri": "{{openAIResourceUri}}",
                "deploymentId": "text-embedding-3-large",
                "apiKey": "{{openAIKey}}",
                "modelName": "text-embedding-3-large"
            }
        }
    ]
},
"semantic": {
    "defaultConfiguration": "semanticconfig",
    "configurations": [
        {
            "name": "semanticconfig",
            "prioritizedFields": {
                "titleField": {
                    "fieldName": "document_title"
                },
                "prioritizedContentFields": [
                ],
                "prioritizedKeywordsFields": []
            }
        }
    ]
}
}
```

Key points:

- Text and image embeddings are stored in the `content_embedding` field and must be configured with appropriate dimensions, such as 3072, and a vector search profile.
- `location_metadata` captures bounding polygon and page number metadata for each text chunk and normalized image, enabling precise spatial search or UI overlays.
- For more information on vector search, see [Vectors in Azure AI Search](#).
- For more information on semantic ranking, see [Semantic ranking in Azure AI Search](#)

Create a skillset

[Create Skillset \(REST\)](#) creates a skillset on your search service. A skillset defines the operations that chunk and embed content prior to indexing. This skillset uses the Document Layout skill to extract text and images, preserving location metadata which is useful for citations in RAG applications. It uses Azure OpenAI Embedding skill to vectorize text content.

The skillset also performs actions specific to images. It uses the GenAI Prompt skill to generate image descriptions. It also creates a knowledge store that stores intact images so that you can return them in a query.

HTTP

```
### Create a skillset
POST {{searchUrl}}/skillsets?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "name": "doc-intelligence-image-verbalization-skillset",
  "description": "A sample skillset for multi-modality using image verbalization",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",
      "name": "document-cracking-skill",
      "description": "Document Layout skill for document cracking",
      "context": "/document",
      "outputMode": "oneToMany",
      "outputFormat": "text",
      "extractionOptions": ["images", "locationMetadata"],
      "chunkingProperties": {
        "unit": "characters",
        "maxLength": 2000,
        "overlapLength": 200
      },
      "inputs": [
        {
          "name": "image"
        }
      ]
    }
  ],
  "knowledgeStore": {
    "name": "knowledge-store-1",
    "type": "Image"
  }
}
```

```
        "name": "file_data",
        "source": "/document/file_data"
    },
],
"outputs": [
{
    "name": "text_sections",
    "targetName": "text_sections"
},
{
    "name": "normalized_images",
    "targetName": "normalized_images"
}
]
},
{
"@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
"name": "text-embedding-skill",
"description": "Azure Open AI Embedding skill for text",
"context": "/document/text_sections/*",
"inputs": [
{
    "name": "text",
    "source": "/document/text_sections/*/content"
}
],
"outputs": [
{
    "name": "embedding",
    "targetName": "text_vector"
}
],
"resourceUri": "{{openAIResourceUri}}",
"deploymentId": "text-embedding-3-large",
"apiKey": "{{openAIKey}}",
"dimensions": 3072,
"modelName": "text-embedding-3-large"
},
{
"@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",
"uri": "{{chatCompletionResourceUri}}",
"timeout": "PT1M",
"apiKey": "{{chatCompletionKey}}",
"name": "genAI-prompt-skill",
"description": "GenAI Prompt skill for image verbalization",
"context": "/document/normalized_images/*",
"inputs": [
{
    "name": "systemMessage",
    "source": "=You are tasked with generating concise, accurate descriptions of images, figures, diagrams, or charts in documents. The goal is to capture the key information and meaning conveyed by the image without including extraneous details like style, colors, visual aesthetics, or size.\n\nInstructions:\nContent Focus: Describe the core content and relationships depicted in the image.\nFor diagrams, specify the main elements and how they are connected or interact.\nFor

```

charts, highlight key data points, trends, comparisons, or conclusions.\nFor figures or technical illustrations, identify the components and their significance.\nClarity & Precision: Use concise language to ensure clarity and technical accuracy. Avoid subjective or interpretive statements.\n\nAvoid Visual Descriptors: Exclude details about:\n\nColors, shading, and visual styles.\nImage size, layout, or decorative elements.\nFonts, borders, and stylistic embellishments.\nContext: If relevant, relate the image to the broader content of the technical document or the topic it supports.\n\nExample Descriptions:\nDiagram: \"A flowchart showing the four stages of a machine learning pipeline: data collection, preprocessing, model training, and evaluation, with arrows indicating the sequential flow of tasks.\"\\n\\nChart: \"A bar chart comparing the performance of four algorithms on three datasets, showing that Algorithm A consistently outperforms the others on Dataset 1.\"\\n\\nFigure: \"A labeled diagram illustrating the components of a transformer model, including the encoder, decoder, self-attention mechanism, and feedforward layers.\"\"

```
    },
    {
      "name": "userMessage",
      "source": "'Please describe this image.'"
    },
    {
      "name": "image",
      "source": "/document/normalized_images/*/data"
    }
  ],
  "outputs": [
    {
      "name": "response",
      "targetName": "verbalizedImage"
    }
  ]
},
{
  "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
  "name": "verbalizedImage-embedding-skill",
  "description": "Azure Open AI Embedding skill for verbalized image embedding",
  "context": "/document/normalized_images/*",
  "inputs": [
    {
      "name": "text",
      "source": "/document/normalized_images/*/verbalizedImage",
      "inputs": []
    }
  ],
  "outputs": [
    {
      "name": "embedding",
      "targetName": "verbalizedImage_vector"
    }
  ],
  "resourceUri": "{{openAIResourceUri}}",
  "deploymentId": "text-embedding-3-large",
  "apiKey": "{{openAIKey}}",
  "dimensions": 3072,
  "modelName": "text-embedding-3-large"
```

```
},
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "#5",
  "context": "/document/normalized_images/*",
  "inputs": [
    {
      "name": "normalized_images",
      "source": "/document/normalized_images/*",
      "inputs": []
    },
    {
      "name": "imagePath",
      "source":
      "my_container_name/'+$(/document/normalized_images/*/imagePath)",
      "inputs": []
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "new_normalized_images"
    }
  ]
},
"indexProjections": {
  "selectors": [
    {
      "targetIndexName": "doc-intelligence-image-verbalization-index",
      "parentKeyFieldName": "text_document_id",
      "sourceContext": "/document/text_sections/*",
      "mappings": [
        {
          "name": "content_embedding",
          "source": "/document/text_sections/*/text_vector"
        },
        {
          "name": "content_text",
          "source": "/document/text_sections/*/content"
        },
        {
          "name": "location_metadata",
          "source": "/document/text_sections/*/locationMetadata"
        },
        {
          "name": "document_title",
          "source": "/document/document_title"
        }
      ]
    },
    {
      "targetIndexName": "doc-intelligence-image-verbalization-index",
      "parentKeyFieldName": "image_document_id",
      "sourceContext": "/document/normalized_images/*",
      "mappings": [
        {
          "name": "image_embedding",
          "source": "/document/normalized_images/*/image_vector"
        }
      ]
    }
  ]
}
```

```

    "mappings": [
        {
            "name": "content_text",
            "source": "/document/normalized_images/*/verbalizedImage"
        },
        {
            "name": "content_embedding",
            "source": "/document/normalized_images/*/verbalizedImage_vector"
        },
        {
            "name": "content_path",
            "source":
                "/document/normalized_images/*/new_normalized_images/imagePath"
        },
        {
            "name": "document_title",
            "source": "/document/document_title"
        },
        {
            "name": "location_metadata",
            "source": "/document/normalized_images/*/locationMetadata"
        }
    ],
    "parameters": {
        "projectionMode": "skipIndexingParentDocuments"
    }
},
"knowledgeStore": {
    "storageConnectionString": "{{storageConnection}}",
    "identity": null,
    "projections": [
        {
            "files": [
                {
                    "storageContainer": "{{imageProjectionContainer}}",
                    "source": "/document/normalized_images/*"
                }
            ]
        }
    ]
}
}

```

This skillset extracts text and images, verbalizes images, and shapes the image metadata for projection into the index.

Key points:

- The `content_text` field is populated in two ways:

- From document text extracted and chunked using the Document Layout skill.
- From image content using the GenAI Prompt skill, which generates descriptive captions for each normalized image
- The `content_embedding` field contains 3072-dimensional embeddings for both page text and verbalized image descriptions. These are generated using the text-embedding-3-large model from Azure OpenAI.
- `content_path` contains the relative path to the image file within the designated image projection container. This field is generated only for images extracted from documents when `extractOption` is set to `["images", "locationMetadata"]` or `["images"]`, and can be mapped from the enriched document from the source field
`/document/normalized_images/*/imagePath.`
- The Azure OpenAI embeddings skill enables embedding of textual data. For more information, see [Azure OpenAI Embedding skill](#).

Create and run an indexer

[Create Indexer](#) creates an indexer on your search service. An indexer connects to the data source, loads data, runs a skillset, and indexes the enriched data.

HTTP

```
### Create and run an indexer
POST {{searchUrl}}/indexers?api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}


{
  "dataSourceName": "doc-intelligence-image-verbalization-ds",
  "targetIndexName": "doc-intelligence-image-verbalization-index",
  "skillsetName": "doc-intelligence-image-verbalization-skillset",
  "parameters": {
    "maxFailedItems": -1,
    "maxFailedItemsPerBatch": 0,
    "batchSize": 1,
    "configuration": {
      "allowSkillsetToReadFileData": true
    }
  },
  "fieldMappings": [
    {
      "sourceFieldName": "metadata_storage_name",
      "targetFieldName": "document_title"
    }
  ],
}
```

```
"outputFieldMappings": []  
}
```

Run queries

You can start searching as soon as the first document is loaded.

HTTP

```
### Query the index  
POST {{searchUrl}}/indexes/doc-intelligence-image-verbalization-index/docs/search?  
api-version=2025-08-01-preview    HTTP/1.1  
Content-Type: application/json  
api-key: {{searchApiKey}}  
  
{  
  "search": "*",  
  "count": true  
}
```

Send the request. This is an unspecified full-text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
{  
  "@odata.count": 100,  
  "@search.nextPageParameters": {  
    "search": "*",  
    "count": true,  
    "skip": 50  
  },  
  "value": [  
  ],  
  "@odata.nextLink": "https://<YOUR-SEARCH-SERVICE-  
NAME>.search.windows.net/indexes/doc-intelligence-image-verbalization-  
index/docs/search?api-version=2025-08-01-preview "  
}
```

100 documents are returned in the response.

For filters, you can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case -sensitive. For more information and examples, see [Examples of simple search queries](#).

(!) Note

The `$filter` parameter only works on fields that were marked filterable during index creation.

HTTP

```
### Query for only images
POST {{searchUrl}}/indexes/doc-intelligence-image-verbalization-index/docs/search?
api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

{
  "search": "*",
  "count": true,
  "filter": "image_document_id ne null"
}
```

HTTP

```
### Query for text or images with content related to energy, returning the id,
parent document, and text (only populated for text chunks), and the content path
where the image is saved in the knowledge store (only populated for images)
POST {{searchUrl}}/indexes/doc-intelligence-image-verbalization-index/docs/search?
api-version=2025-08-01-preview    HTTP/1.1
Content-Type: application/json
api-key: {{searchApiKey}}

{
  "search": "energy",
  "count": true,
  "select": "content_id, document_title, content_text, content_path"
}
```

Reset and rerun

Indexers can be reset to clear execution history, which allows a full rerun. The following POST requests are for reset, followed by rerun.

HTTP

```
### Reset the indexer
POST {{searchUrl}}/indexers/doc-intelligence-image-verbalization-indexer/reset?
api-version=2025-08-01-preview    HTTP/1.1
api-key: {{searchApiKey}}
```

```
HTTP
```

```
### Run the indexer
POST {{searchUrl}}/indexers/doc-intelligence-image-verbalization-indexer/run?api-version=2025-08-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

```
HTTP
```

```
### Check indexer status
GET {{searchUrl}}/indexers/doc-intelligence-image-verbalization-indexer/status?
api-version=2025-08-01-preview HTTP/1.1
    api-key: {{searchApiKey}}
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can use the Azure portal to delete indexes, indexers, and data sources.

See also

Now that you're familiar with a sample implementation of a multimodal indexing scenario, check out:

- [GenAI Prompt skill](#)
- [Document Layout skill](#)
- [Azure OpenAI Embedding skill](#)
- [Vectors in Azure AI Search](#)
- [Semantic ranking in Azure AI Search](#)

Tutorial: Build an end-to-end agentic retrieval solution using Azure AI Search

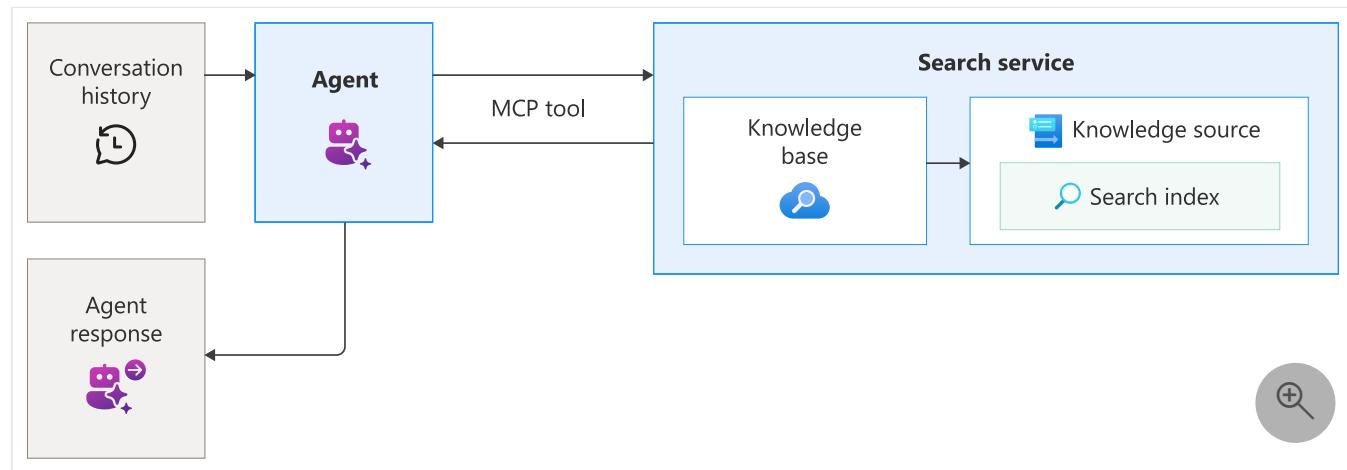
! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this tutorial, you learn how to build a solution that integrates Azure AI Search and Foundry Agent Service for intelligent knowledge retrieval.

This solution uses Model Context Protocol (MCP) to establish a standardized connection between your agentic retrieval pipeline in Azure AI Search, which consists of a *knowledge base* that references a *knowledge source*, and your agent in Foundry Agent Service. You can use this architecture for conversational applications that require complex reasoning over large knowledge domains, such as customer support or technical troubleshooting.

The following diagram shows the high-level architecture of this agentic retrieval solution:



💡 Tip

- Want to get started right away? See the [agentic-retrieval-pipeline-example](#) source code.
- Want a simpler introduction to agentic retrieval? See [Quickstart: Use agentic retrieval](#).

Prerequisites

- An Azure AI Search service in any region that provides agentic retrieval.
- A [Microsoft Foundry project](#) and resource. When you create a project, the resource is automatically created.
- A [supported LLM](#) deployed to your project. We recommend a minimum token capacity of 100,000. You can find the LLM's capacity and rate limit in the Foundry portal. If you want [vectorization at query time](#), you should also deploy a text embedding model.
- [Authentication and permissions](#) on your search service and project.
- Preview package versions. For a complete list of versions used in this solution, see the [requirements.txt](#) file.

Authentication and permissions

Before you begin, make sure you have permissions to access content and operations. We recommend Microsoft Entra ID authentication and role-based access for authorization. You must be an **Owner** or **User Access Administrator** to assign roles. If roles aren't feasible, you can use [key-based authentication](#) instead.

To configure access for this solution, select both of the following tabs.

Azure AI Search

1. [Enable role-based access.](#)
2. [Configure a managed identity.](#)
3. [Assign roles:](#)
 - You must have the **Search Service Contributor**, **Search Index Data Contributor**, and **Search Index Data Reader** roles to create, load, and retrieve on Azure AI Search.
 - For integrated operations, ensure that all clients using the retrieval pipeline have the **Search Index Data Reader** role for sending retrieval requests.

Understand the solution

This section pairs each component of the solution with its corresponding development tasks. For deeper guidance, see the linked how-to articles.

Azure AI Search

Azure AI Search hosts your indexed content and the agentic retrieval pipeline.

Development tasks include:

- Create a [knowledge source](#). Agentic retrieval supports multiple types of knowledge sources, but this solution creates a [search index knowledge source](#).
- [Create a knowledge base](#) that maps to your LLM deployment and uses the extractive data output mode. We recommend this output mode for interaction with Foundry Agent Service because it provides the agent with verbatim, unprocessed content for grounding and reasoning.

A user initiates query processing by interacting with a client app, such as a chatbot, that calls the agent. The agent uses the MCP tool to orchestrate requests to the knowledge base and synthesize responses. When the chatbot calls the agent, the MCP tool calls the knowledge base in Azure AI Search and sends it back to the agent and chatbot.

Build the solution

Follow these steps to create an end-to-end agentic retrieval solution.

Get endpoints

For this solution, you need the following endpoints:

Azure AI Search

- The endpoint for your search service, which you can find on the [Overview](#) page in the Azure portal. It should look like this: `https://{your-service-name}.search.windows.net/`

Create agentic retrieval objects

This section omits code snippets for creating the knowledge source and knowledge base in Azure AI Search, skipping ahead to the Foundry Agent Service integration. For more

information about the omitted steps, see the [Understand the solution](#) section.

Create a project connection

Before you can use the MCP tool in an agent, you must create a project connection in Foundry that points to the `mcp_endpoint` of your knowledge base. This endpoint allows the agent to access your knowledge base.

Python

```
import requests
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

# Provide connection details
credential = DefaultAzureCredential()
project_resource_id = "{project_resource_id}" # e.g.
/subscriptions/{subscription}/resourceGroups/{resource_group}/providers/Microsoft.Ma
chineLearningServices/workspaces/{account_name}/projects/{project_name}
project_connection_name = "{project_connection_name}"
mcp_endpoint = "{search_service_endpoint}/knowledgebases/{knowledge_base_name}/mcp?
api-version=2025-11-01-preview" # This endpoint enables the MCP connection between
the agent and knowledge base

# Get bearer token for authentication
bearer_token_provider = get_bearer_token_provider(credential,
"https://management.azure.com/.default")
headers = {
    "Authorization": f"Bearer {bearer_token_provider()}",

}

# Create project connection
response = requests.put(
    f"https://management.azure.com{project_resource_id}/connections/{project_connection_
name}?api-version=2025-10-01-preview",
    headers = headers,
    json = {
        "name": "project_connection_name",
        "type": "Microsoft.MachineLearningServices/workspaces/connections",
        "properties": {
            "authType": "ProjectManagedIdentity",
            "category": "RemoteTool",
            "target": mcp_endpoint,
            "isSharedToAll": True,
            "audience": "https://search.azure.com/",
            "metadata": { "ApiType": "Azure" }
        }
    }
)
```

```
response.raise_for_status()
print(f"Connection '{project_connection_name}' created or updated successfully.")
```

Set up an AI project client

Use [AIProjectClient](#) to create a client connection to your Foundry project.

Python

```
from azure.ai.projects import AIProjectClient

project_client = AIProjectClient(endpoint=project_endpoint, credential=credential)

list(project_client.agents.list())
```

Create an agent that uses the MCP tool

The next step is to create an agent configured with the MCP tool. When the agent receives a user query, it can call your knowledge base through the MCP tool to retrieve relevant content for response grounding.

The agent definition includes instructions that specify its behavior and the project connection you previously created. For more information, see [Quickstart: Create a new agent](#).

Python

```
from azure.ai.projects.models import PromptAgentDefinition, MCPTool

# Define agent instructions
instructions = """
A Q&A agent that can answer questions based on the attached knowledge base.
Always provide references to the ID of the data source used to answer the question.
If you don't have the answer, respond with "I don't know".
"""

# Create MCP tool with knowledge base connection
mcp_kb_tool = MCPTool(
    server_label = "knowledge-base",
    server_url = mcp_endpoint,
    require_approval = "never",
    allowed_tools = ["knowledge_base_retrieve"],
    project_connection_id = project_connection_name
)

# Create agent with MCP tool
agent = project_client.agents.create_version(
    agent_name = agent_name,
    definition = PromptAgentDefinition(
        model = agent_model,
```

```
        instructions = instructions,
        tools = [mcp_kb_tool]
    )

print(f"Agent '{agent_name}' created or updated successfully.")
```

Chat with the agent

Your client app uses the Conversations and [Responses](#) APIs from Azure OpenAI to send user input to the agent. The client creates a conversation and passes each user message to the agent through the Responses API, resembling a typical chat experience.

The agent manages the conversation, determines when to call your knowledge base through the MCP tool, and returns a natural-language response (with references to the retrieved content) to the client app.

Python

```
# Get the OpenAI client for responses and conversations
openai_client = project_client.get_openai_client()

# Create conversation
conversation = openai_client.conversations.create()

# Send request to trigger the MCP tool
response = openai_client.responses.create(
    conversation = conversation.id,
    input = """
        Why do suburban belts display larger December brightening than urban cores
        even though absolute light levels are higher downtown?
        Why is the Phoenix nighttime street grid is so sharply visible from space,
        whereas large stretches of the interstate between midwestern cities remain
        comparatively dim?
        """,
    extra_body = {"agent": {"name": agent.name, "type": "agent_reference"}},
)

print(f"Response: {response.output_text}")
```

Improve data quality

By default, search results from your knowledge base are consolidated into a large unified string that can be passed to the agent for grounding. Azure AI Search provides the following indexing and relevance-tuning features to help you generate high-quality results. You can implement these features in the search index, and the improvements in search relevance are evident in the quality of retrieval responses.

- [Scoring profiles](#) provide built-in boosting criteria. Your index must specify a default scoring profile, which is used by the retrieval engine when queries include fields associated with that profile.
- [Semantic configuration](#) is required, but you determine which fields are prioritized and used for ranking.
- For plain-text content, you can use [analyzers](#) to control tokenization during indexing.
- For [multimodal or image content](#), you can use image verbalization for LLM-generated descriptions of your images or classic OCR and image analysis via skillsets during indexing.

Control the number of subqueries

The LLM that powers your knowledge base determines the number of subqueries based on the following factors:

- User query
- Chat history
- Semantic ranker input constraints

As the developer, you can control the number of subqueries by [setting the retrieval reasoning effort](#). The reasoning effort determines the level of LLM processing for query planning, ranging from minimal (no LLM processing) to medium (deeper search and follow-up iterations).

Control the context sent to the agent

The Responses API controls what is sent to the agent and knowledge base. To optimize performance and relevance, adjust your agent instructions to summarize or filter the chat history before sending it to the MCP tool.

Control costs and limit operations

For insights into the query plan, look at output tokens in the [activity array](#) of knowledge base responses.

Improve performance

To optimize performance and reduce latency, consider the following strategies:

- Summarize message threads.
- Use `gpt-4.1-mini` or a smaller model that performs faster.
- Set `maxOutputSize` on the [retrieve action](#) to govern the size of the response or `maxRuntimeInSeconds` for time-bound processing.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can also delete individual objects:

Python

```
# Delete the agent
project_client.agents.delete_version(agent.name, agent.version)
print(f"AI agent '{agent.name}' version '{agent.version}' deleted successfully")

# Delete the knowledge base
index_client.delete_knowledge_base(base_name)
print(f"Knowledge base '{base_name}' deleted successfully")

# Delete the knowledge source
index_client.delete_knowledge_source(knowledge_source=knowledge_source_name) # This
is new feature in 2025-08-01-Preview api version
print(f"Knowledge source '{knowledge_source_name}' deleted successfully.")

# Delete the search index
index_client.delete_index(index)
print(f"Index '{index_name}' deleted successfully")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Tutorial: Skillsets in Azure AI Search

Learn how to use the [Azure SDK for .NET](#) to create an [AI enrichment pipeline](#) for content extraction and transformations during indexing.

Skillsets add AI processing to raw content, making it more uniform and searchable. Once you know how skillsets work, you can support a broad range of transformations, from image analysis to natural language processing to customized processing that you provide externally.

In this tutorial, you:

- ✓ Define objects in an enrichment pipeline.
- ✓ Build a skillset. Invoke OCR, language detection, entity recognition, and key phrase extraction.
- ✓ Execute the pipeline. Create and load a search index.
- ✓ Check the results using full-text search.

Overview

This tutorial uses C# and the [Azure.Search.Documents](#) client library to create a data source, index, indexer, and skillset.

The [indexer](#) drives each step in the pipeline, starting with content extraction of sample data (unstructured text and images) in a blob container on Azure Storage.

Once content is extracted, the [skillset](#) executes built-in skills from Microsoft to find and extract information. These skills include Optical Character Recognition (OCR) on images, language detection on text, key phrase extraction, and entity recognition (organizations). New information created by the skillset is sent to fields in an [index](#). Once the index is populated, you can use the fields in queries, facets, and filters.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Azure Storage](#).
- [Azure AI Search](#).
- [Azure.Search.Documents package](#).
- [Visual Studio](#).

(!) Note

You can use a free search service for this tutorial. The Free tier limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before you start, make sure you have room on your service to accept the new resources.

Download files

- Download a zip file of the sample data repository and extract the contents. [Learn how ↗](#).
- Download the [sample data files \(mixed media\) ↗](#)

Upload sample data to Azure Storage

1. In Azure Storage, [create a new container](#) and name it *mixed-content-types*.
2. Upload the sample data files.
3. Get a storage connection string so that you can formulate a connection in Azure AI Search.
 - a. On the left, select **Access keys**.
 - b. Copy the connection string for either key one or key two. The connection string is similar to the following example:

```
DefaultEndpointsProtocol=https;AccountName=<your account name>;AccountKey=<your  
account key>;EndpointSuffix=core.windows.net
```

Foundry Tools

Built-in AI enrichment is backed by Foundry Tools, including Azure Language and Azure Vision for natural language and image processing. For small workloads like this tutorial, you can use the free allocation of 20 transactions per indexer. For larger workloads, [attach a Microsoft Foundry resource to a skillset](#) for Standard pricing.

Copy a search service URL and API key

For this tutorial, connections to Azure AI Search require an endpoint and an API key. You can get these values from the Azure portal.

1. Sign in to the [Azure portal ↗](#) and select your search service.

2. From the left pane, select **Overview** and copy the endpoint. It should be in this format:

`https://my-service.search.windows.net`

3. From the left pane, select **Settings > Keys** and copy an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either key on requests to add, modify, or delete objects.

Set up your environment

Begin by opening Visual Studio and creating a new Console App project.

Install Azure.Search.Documents

The [Azure AI Search .NET SDK](#) consists of a client library that enables you to manage your indexes, data sources, indexers, and skillsets, as well as upload and manage documents and execute queries, all without having to deal with the details of HTTP and JSON. This client library is distributed as a NuGet package.

For this project, install version 11 or later of the `Azure.Search.Documents` and the latest version of `Microsoft.Extensions.Configuration`.

1. In Visual Studio, select **Tools > NuGet Package Manager > Manage NuGet Packages for Solution...**
2. Browse for [Azure.Search.Document](#).
3. Select the latest version and then select **Install**.
4. Repeat the previous steps to install [Microsoft.Extensions.Configuration](#) and [Microsoft.Extensions.Configuration.Json](#).

Add service connection information

1. Right-click on your project in the Solution Explorer and select **Add > New Item...**
2. Name the file `appsettings.json` and select **Add**.
3. Include this file in your output directory.
 - a. Right-click on `appsettings.json` and select **Properties**.
 - b. Change the value of **Copy to Output Directory** to **Copy if newer**.
4. Copy the below JSON into your new JSON file.

JSON

```
{  
    "SearchServiceUri": "<YourSearchServiceUri>",  
    "SearchServiceAdminApiKey": "<YourSearchServiceAdminApiKey>",  
    "SearchServiceQueryApiKey": "<YourSearchServiceQueryApiKey>",  
    "AzureAISServicesKey": "<YourMultiRegionAzureAISServicesKey>",  
    "AzureBlobConnectionString": "<YourAzureBlobConnectionString>"  
}
```

Add your search service and blob storage account information. Recall that you can get this information from the service provisioning steps indicated in the previous section.

For **SearchServiceUri**, enter the full URL.

Add namespaces

In `Program.cs`, add the following namespaces.

C#

```
using Azure;  
using Azure.Search.Documents.Indexes;  
using Azure.Search.Documents.Indexes.Models;  
using Microsoft.Extensions.Configuration;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
  
namespace EnrichwithAI
```

Create a client

Create an instance of a `SearchIndexClient` and a `SearchIndexerClient` under `Main`.

C#

```
public static void Main(string[] args)  
{  
    // Create service client  
    IConfigurationBuilder builder = new  
    ConfigurationBuilder().AddJsonFile("appsettings.json");  
    IConfigurationRoot configuration = builder.Build();  
  
    string searchServiceUri = configuration["SearchServiceUri"];  
    string adminApiKey = configuration["SearchServiceAdminApiKey"];  
    string azureAiServicesKey = configuration["AzureAISServicesKey"];  
  
    SearchIndexClient indexClient = new SearchIndexClient(new Uri(searchServiceUri),
```

```
new AzureKeyCredential(adminApiKey));
    SearchIndexerClient indexerClient = new SearchIndexerClient(new
Uri(searchServiceUri), new AzureKeyCredential(adminApiKey));
}
```

! Note

The clients connect to your search service. In order to avoid opening too many connections, you should try to share a single instance in your application if possible. The methods are thread-safe to enable such sharing.

Add a function to exit the program during failure

This tutorial is meant to help you understand each step of the indexing pipeline. If there's a critical issue that prevents the program from creating the data source, skillset, index, or indexer the program will output the error message and exit so that the issue can be understood and addressed.

Add `ExitProgram` to `Main` to handle scenarios that require the program to exit.

C#

```
private static void ExitProgram(string message)
{
    Console.WriteLine("{0}", message);
    Console.WriteLine("Press any key to exit the program...");
    Console.ReadKey();
    Environment.Exit(0);
}
```

Create the pipeline

In Azure AI Search, AI processing occurs during indexing (or data ingestion). This part of the walkthrough creates four objects: data source, index definition, skillset, indexer.

Step 1: Create a data source

`SearchIndexerClient` has a `DataSourceName` property that you can set to a `SearchIndexerDataSourceConnection` object. This object provides all the methods you need to create, list, update, or delete Azure AI Search data sources.

Create a new `SearchIndexerDataSourceConnection` instance by calling `indexerClient.CreateOrUpdateDataSourceConnection(dataSource)`. The following code creates a data source of type `AzureBlob`.

C#

```
private static SearchIndexerDataSourceConnection
CreateOrUpdateDataSource(SearchIndexerClient indexerClient, IConfigurationRoot
configuration)
{
    SearchIndexerDataSourceConnection dataSource = new
    SearchIndexerDataSourceConnection(
        name: "demodata",
        type: SearchIndexerDataSourceType.AzureBlob,
        connectionString: configuration["AzureBlobConnectionString"],
        container: new SearchIndexerDataContainer("mixed-content-type"))
    {
        Description = "Demo files to demonstrate Azure AI Search capabilities."
    };

    // The data source does not need to be deleted if it was already created
    // since we are using the CreateOrUpdate method
    try
    {
        indexerClient.CreateOrUpdateDataSourceConnection(dataSource);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed to create or update the data source\n Exception
message: {0}\n", ex.Message);
        ExitProgram("Cannot continue without a data source");
    }

    return dataSource;
}
```

For a successful request, the method returns the data source that was created. If there's a problem with the request, such as an invalid parameter, the method throws an exception.

Now add a line in `Main` to call the `CreateOrUpdateDataSource` function that you've just added.

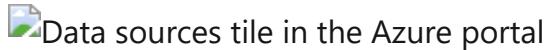
C#

```
// Create or Update the data source
Console.WriteLine("Creating or updating the data source...");
```

```
SearchIndexerDataSourceConnection dataSource =
CreateOrUpdateDataSource(indexerClient, configuration);
```

Build and run the solution. Since this is your first request, check the Azure portal to confirm the data source was created in Azure AI Search. On the search service overview page, verify the

Data Sources list has a new item. You might need to wait a few minutes for the Azure portal page to refresh.



Step 2: Create a skillset

In this section, you define a set of enrichment steps that you want to apply to your data. Each enrichment step is called a *skill* and the set of enrichment steps, a *skillset*. This tutorial uses [built-in skills](#) for the skillset:

- [Optical Character Recognition](#) to recognize printed and handwritten text in image files.
- [Text Merger](#) to consolidate text from a collection of fields into a single "merged content" field.
- [Language Detection](#) to identify the content's language.
- [Entity Recognition](#) for extracting the names of organizations from content in the blob container.
- [Text Split](#) to break large content into smaller chunks before calling the key phrase extraction skill and the entity recognition skill. Key phrase extraction and entity recognition accept inputs of 50,000 characters or less. A few of the sample files need splitting up to fit within this limit.
- [Key Phrase Extraction](#) to pull out the top key phrases.

During initial processing, Azure AI Search cracks each document to extract content from different file formats. Text originating in the source file is placed into a generated `content` field, one for each document. As such, set the input as `"/document/content"` to use this text. Image content is placed into a generated `normalized_images` field, specified in a skillset as `/document/normalized_images/*`.

Outputs can be mapped to an index, used as input to a downstream skill, or both as is the case with language code. In the index, a language code is useful for filtering. As an input, language code is used by text analysis skills to inform the linguistic rules around word breaking.

For more information about skillset fundamentals, see [How to define a skillset](#).

OCR skill

The [OcrSkill](#) extracts text from images. This skill assumes that a `normalized_images` field exists. To generate this field, later in the tutorial we set the `"imageAction"` configuration in the indexer

definition to "generateNormalizedImages".

C#

```
private static OcrSkill CreateOcrSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("image")
    {
        Source = "/document/normalized_images/*"
    });

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("text")
    {
        TargetName = "text"
    });

    OcrSkill ocrSkill = new OcrSkill(inputMappings, outputMappings)
    {
        Description = "Extract text (plain and structured) from image",
        Context = "/document/normalized_images/*",
        DefaultLanguageCode = OcrSkillLanguage.En,
        ShouldDetectOrientation = true
    };

    return ocrSkill;
}
```

Merge skill

In this section, you create a [MergeSkill](#) that merges the document content field with the text that was produced by the OCR skill.

C#

```
private static MergeSkill CreateMergeSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/content"
    });
    inputMappings.Add(new InputFieldMappingEntry("itemsToInsert")
    {
        Source = "/document/normalized_images/*/text"
    });
    inputMappings.Add(new InputFieldMappingEntry("offsets")
    {
        Source = "/document/normalized_images/*/contentOffset"
    });
}
```

```

    });

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("mergedText")
    {
        TargetName = "merged_text"
    });

    MergeSkill mergeSkill = new MergeSkill(inputMappings, outputMappings)
    {
        Description = "Create merged_text which includes all the textual representation of each image inserted at the right location in the content field.",
        Context = "/document",
        InsertPreTag = " ",
        InsertPostTag = " "
    };

    return mergeSkill;
}

```

Language detection skill

The [LanguageDetectionSkill](#) detects the language of the input text and reports a single language code for every document submitted on the request. We use the output of the **Language Detection** skill as part of the input to the **Text Split** skill.

C#

```

private static LanguageDetectionSkill CreateLanguageDetectionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/merged_text"
    });

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("languageCode")
    {
        TargetName = "languageCode"
    });

    LanguageDetectionSkill languageDetectionSkill = new
    LanguageDetectionSkill(inputMappings, outputMappings)
    {
        Description = "Detect the language used in the document",
        Context = "/document"
    };
}

```

```
    return languageDetectionSkill;
}
```

Text split skill

The below [SplitSkill](#) splits text by pages and limits the page length to 4,000 characters as measured by `String.Length`. The algorithm tries to split the text into chunks that are at most `maximumPageLength` in size. In this case, the algorithm does its best to break the sentence on a sentence boundary, so the size of the chunk might be slightly less than `maximumPageLength`.

C#

```
private static SplitSkill CreateSplitSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/merged_text"
    });
    inputMappings.Add(new InputFieldMappingEntry("languageCode")
    {
        Source = "/document/languageCode"
    });

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("textItems")
    {
        TargetName = "pages",
    });

    SplitSkill splitSkill = new SplitSkill(inputMappings, outputMappings)
    {
        Description = "Split content into pages",
        Context = "/document",
        TextSplitMode = TextSplitMode.Pages,
        MaximumPageLength = 4000,
        DefaultLanguageCode = SplitSkillLanguage.English
    };

    return splitSkill;
}
```

Entity recognition skill

This `EntityRecognitionSkill` instance is set to recognize category type `organization`. The `EntityRecognitionSkill` can also recognize category types `person` and `location`.

Notice that the "context" field is set to `"/document/pages/*"` with an asterisk, meaning the enrichment step is called for each page under `"/document/pages"`.

C#

```
private static EntityRecognitionSkill CreateEntityRecognitionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/pages/*"
    });

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("organizations")
    {
        TargetName = "organizations"
    });

    // Specify the V3 version of the EntityRecognitionSkill
    var skillVersion = EntityRecognitionSkill.SkillVersion.V3;

    var entityRecognitionSkill = new EntityRecognitionSkill(inputMappings,
    outputMappings, skillVersion)
    {
        Description = "Recognize organizations",
        Context = "/document/pages/*",
        DefaultLanguageCode = EntityRecognitionSkillLanguage.English
    };
    entityRecognitionSkill.Categories.Add(EntityCategory.Organization);
    return entityRecognitionSkill;
}
```

Key phrase extraction skill

Like the `EntityRecognitionSkill` instance that was just created, the `KeyPhraseExtractionSkill` is called for each page of the document.

C#

```
private static KeyPhraseExtractionSkill CreateKeyPhraseExtractionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/pages/*"
    });
    inputMappings.Add(new InputFieldMappingEntry("languageCode")
    {
```

```

        Source = "/document/languageCode"
    });

List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>()
();
outputMappings.Add(new OutputFieldMappingEntry("keyPhrases")
{
    TargetName = "keyPhrases"
});

KeyPhraseExtractionSkill keyPhraseExtractionSkill = new
KeyPhraseExtractionSkill(inputMappings, outputMappings)
{
    Description = "Extract the key phrases",
    Context = "/document/pages/*",
    DefaultLanguageCode = KeyPhraseExtractionSkillLanguage.En
};

return keyPhraseExtractionSkill;
}

```

Build and create the skillset

Build the [SearchIndexerSkillset](#) using the skills you created.

C#

```

private static SearchIndexerSkillset CreateOrUpdateDemoSkillSet(SearchIndexerClient
indexerClient, IList<SearchIndexerSkill> skills, string azureAiServicesKey)
{
    SearchIndexerSkillset skillset = new SearchIndexerSkillset("demoskillset",
skills)
    {
        // Foundry Tools was formerly known as Cognitive Services.
        // The APIs still use the old name, so we need to create a
CognitiveServicesAccountKey object.
        Description = "Demo skillset",
        CognitiveServicesAccount = new
CognitiveServicesAccountKey(azureAiServicesKey)
    };

    // Create the skillset in your search service.
    // The skillset does not need to be deleted if it was already created
    // since we are using the CreateOrUpdate method
    try
    {
        indexerClient.CreateOrUpdateSkillset(skillset);
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to create the skillset\n Exception message:
{0}\n", ex.Message);
    }
}

```

```

        ExitProgram("Cannot continue without a skillset");
    }

    return skillset;
}

```

Add the following lines to `Main`.

C#

```

// Create the skills
Console.WriteLine("Creating the skills...");
OcrSkill ocrSkill = CreateOcrSkill();
MergeSkill mergeSkill = CreateMergeSkill();
EntityRecognitionSkill entityRecognitionSkill = CreateEntityRecognitionSkill();
LanguageDetectionSkill languageDetectionSkill = CreateLanguageDetectionSkill();
SplitSkill splitSkill = CreateSplitSkill();
KeyPhraseExtractionSkill keyPhraseExtractionSkill =
CreateKeyPhraseExtractionSkill();

// Create the skillset
Console.WriteLine("Creating or updating the skillset...");
List<SearchIndexerSkill> skills = new List<SearchIndexerSkill>();
skills.Add(ocrSkill);
skills.Add(mergeSkill);
skills.Add(languageDetectionSkill);
skills.Add(splitSkill);
skills.Add(entityRecognitionSkill);
skills.Add(keyPhraseExtractionSkill);

SearchIndexerSkillset skillset = CreateOrUpdateDemoSkillSet(indexerClient, skills,
azureAiServicesKey);

```

Step 3: Create an index

In this section, you define the index schema by specifying which fields to include in the searchable index, and the search attributes for each field. Fields have a type and can take attributes that determine how the field is used (searchable, sortable, and so forth). Field names in an index aren't required to identically match the field names in the source. In a later step, you add field mappings in an indexer to connect source-destination fields. For this step, define the index using field naming conventions pertinent to your search application.

This exercise uses the following fields and field types:

[] Expand table

Field names	Field types
id	Edm.String
content	Edm.String
languageCode	Edm.String
keyPhrases	List<Edm.String>
organizations	List<Edm.String>

Create DemoIndex Class

The fields for this index are defined using a model class. Each property of the model class has attributes that determine the search-related behaviors of the corresponding index field.

We'll add the model class to a new C# file. Right select on your project and select **Add > New Item...**, select "Class" and name the file `DemoIndex.cs`, then select **Add**.

Make sure to indicate that you want to use types from the `Azure.Search.Documents.Indexes` and `System.Text.Json.Serialization` namespaces.

Add the below model class definition to `DemoIndex.cs` and include it in the same namespace where you create the index.

C#

```
using Azure.Search.Documents.Indexes;
using System.Text.Json.Serialization;

namespace EnrichwithAI
{
    // The SerializePropertyNamesAsCamelCase is currently unsupported as of this
    // writing.
    // Replace it with JsonPropertyName
    public class DemoIndex
    {
        [SearchableField(IsSortable = true, IsKey = true)]
        [JsonPropertyName("id")]
        public string Id { get; set; }

        [SearchableField]
        [JsonPropertyName("content")]
        public string Content { get; set; }

        [SearchableField]
        [JsonPropertyName("languageCode")]
        public string LanguageCode { get; set; }
    }
}
```

```

[SearchableField]
[JsonPropertyName("keyPhrases")]
public string[] KeyPhrases { get; set; }

[SearchableField]
[JsonPropertyName("organizations")]
public string[] Organizations { get; set; }
}

}

```

Now that you've defined a model class, back in `Program.cs` you can create an index definition fairly easily. The name for this index will be `demoindex`. If an index already exists with that name, it's deleted.

C#

```

private static SearchIndex CreateDemoIndex(SearchIndexClient indexClient)
{
    FieldBuilder builder = new FieldBuilder();
    var index = new SearchIndex("demoindex")
    {
        Fields = builder.Build(typeof(DemoIndex))
    };

    try
    {
        indexClient.GetIndex(index.Name);
        indexClient.DeleteIndex(index.Name);
    }
    catch (RequestFailedException ex) when (ex.Status == 404)
    {
        //if the specified index not exist, 404 will be thrown.
    }

    try
    {
        indexClient.CreateIndex(index);
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to create the index\n Exception message: {0}\n",
ex.Message);
        ExitProgram("Cannot continue without an index");
    }

    return index;
}

```

During testing, you might find that you're attempting to create the index more than once. Because of this, check to see if the index that you're about to create already exists before

attempting to create it.

Add the following lines to `Main`.

```
C#
```

```
// Create the index
Console.WriteLine("Creating the index...");
SearchIndex demoIndex = CreateDemoIndex(indexClient);
```

Add the following using statement to resolve the disambiguated reference.

```
C#
```

```
using Index = Azure.Search.Documents.Indexes.Models;
```

To learn more about index concepts, see [Create Index \(REST API\)](#).

Step 4: Create and run an indexer

So far you have created a data source, a skillset, and an index. These three components become part of an [indexer](#) that pulls each piece together into a single multi-phased operation. To tie these together in an indexer, you must define field mappings.

- The `fieldMappings` are processed before the skillset, mapping source fields from the data source to target fields in an index. If field names and types are the same at both ends, no mapping is required.
- The `outputFieldMappings` are processed after the skillset, referencing `sourceFieldNames` that don't exist until document cracking or enrichment creates them. The `targetFieldName` is a field in an index.

In addition to hooking up inputs to outputs, you can also use field mappings to flatten data structures. For more information, see [How to map enriched fields to a searchable index](#).

```
C#
```

```
private static SearchIndexer CreateDemoIndexer(SearchIndexerClient indexerClient,
SearchIndexerDataSourceConnection dataSource, SearchIndexerSkillset skillSet,
SearchIndex index)
{
    IndexingParameters indexingParameters = new IndexingParameters()
    {
        MaxFailedItems = -1,
        MaxFailedItemsPerBatch = -1,
    };
    indexingParameters.Configuration.Add("dataToExtract", "contentAndMetadata");
```

```
indexingParameters.Configuration.Add("imageAction", "generateNormalizedImages");

SearchIndexer indexer = new SearchIndexer("demoindexer", dataSource.Name,
index.Name)
{
    Description = "Demo Indexer",
    SkillsetName = skillSet.Name,
    Parameters = indexingParameters
};

FieldMappingFunction mappingFunction = new FieldMappingFunction("base64Encode");
mappingFunction.Parameters.Add("useHttpServerUtilityUrlTokenEncode", true);

indexer.FieldMappings.Add(new FieldMapping("metadata_storage_path")
{
    TargetFieldName = "id",
    MappingFunction = mappingFunction
});

indexer.FieldMappings.Add(new FieldMapping("content")
{
    TargetFieldName = "content"
});

indexer.OutputFieldMappings.Add(new
FieldMapping("/document/pages/*/organizations/*")
{
    TargetFieldName = "organizations"
});
indexer.OutputFieldMappings.Add(new
FieldMapping("/document/pages/*/keyPhrases/*")
{
    TargetFieldName = "keyPhrases"
});
indexer.OutputFieldMappings.Add(new FieldMapping("/document/languageCode")
{
    TargetFieldName = "languageCode"
});

try
{
    indexerClient.GetIndexer(indexer.Name);
    indexerClient.DeleteIndexer(indexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    //if the specified indexer not exist, 404 will be thrown.
}

try
{
    indexerClient.CreateIndexer(indexer);
}
catch (RequestFailedException ex)
{
```

```
        Console.WriteLine("Failed to create the indexer\n Exception message: {0}\n",  
    ex.Message);  
        ExitProgram("Cannot continue without creating an indexer");  
    }  
  
    return indexer;  
}
```

Add the following lines to `Main`.

C#

```
// Create the indexer, map fields, and execute transformations  
Console.WriteLine("Creating the indexer and executing the pipeline...");  
SearchIndexer demoIndexer = CreateDemoIndexer(indexerClient, dataSource, skillset,  
demoIndex);
```

Expect indexer processing to take some time to complete. Even though the data set is small, analytical skills are computation-intensive. Some skills, such as image analysis, are long-running.

💡 Tip

Creating an indexer invokes the pipeline. If there are problems reaching the data, mapping inputs and outputs, or order of operations, they appear at this stage.

Explore creating the indexer

The code sets `"maxFailedItems"` to -1, which instructs the indexing engine to ignore errors during data import. This is useful because there are so few documents in the demo data source. For a larger data source, you would set the value to greater than 0.

Also notice the `"dataToExtract"` is set to `"contentAndMetadata"`. This statement tells the indexer to automatically extract the content from different file formats as well as metadata related to each file.

When content is extracted, you can set `imageAction` to extract text from images found in the data source. The `"imageAction"` set to `"generateNormalizedImages"` configuration, combined with the OCR Skill and Text Merge Skill, tells the indexer to extract text from the images (for example, the word "stop" from a traffic Stop sign), and embed it as part of the content field. This behavior applies to both the images embedded in the documents (think of an image inside a PDF), as well as images found in the data source, for instance a JPG file.

Monitor indexing

Once the indexer is defined, it runs automatically when you submit the request. Depending on which skills you defined, indexing can take longer than you expect. To find out whether the indexer is still running, use the `GetStatus` method.

C#

```
private static void CheckIndexerOverallStatus(SearchIndexerClient indexerClient,
SearchIndexer indexer)
{
    try
    {
        var demoIndexerExecutionInfo = indexerClient.GetIndexerStatus(indexer.Name);

        switch (demoIndexerExecutionInfo.Value.Status)
        {
            case IndexerStatus.Error:
                ExitProgram("Indexer has error status. Check the Azure portal to
further understand the error.");
                break;
            case IndexerStatus.Running:
                Console.WriteLine("Indexer is running");
                break;
            case IndexerStatus.Unknown:
                Console.WriteLine("Indexer status is unknown");
                break;
            default:
                Console.WriteLine("No indexer information");
                break;
        }
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to get indexer overall status\n Exception message:
{0}\n", ex.Message);
    }
}
```

`demoIndexerExecutionInfo` represents the current status and execution history of an indexer.

Warnings are common with some source file and skill combinations and don't always indicate a problem. In this tutorial, the warnings are benign (for example, no text inputs from the JPEG files).

Add the following lines to `Main`.

C#

```
// Check indexer overall status
Console.WriteLine("Check the indexer overall status...");
CheckIndexerOverallStatus(indexerClient, demoIndexer);
```

Search

In Azure AI Search tutorial console apps, we typically add a 2-second delay before running queries that return results, but because enrichment takes several minutes to complete, we'll close the console app and use another approach instead.

The easiest option is [Search Explorer](#) in the Azure portal. You can first run an empty query that returns all documents, or a more targeted search that returns new field content created by the pipeline.

1. In the Azure portal, in the search service pages, expand **Search Management > Indexes**.
2. Find **demoindex** in the list. It should have 14 documents. If the document count is zero, the indexer is either still running or the page hasn't been refreshed yet.
3. Select **demoindex**. Search Explorer is the first tab.
4. Content is searchable as soon as the first document is loaded. To verify content exists, run an unspecified query by clicking **Search**. This query returns all currently indexed documents, giving you an idea of what the index contains.
5. For more manageable results, switch to JSON view and set parameters to select the fields:

JSON

```
{
  "search": "*",
  "count": true,
  "select": "id, languageCode, organizations"
}
```

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure AI Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing objects and deletes them so that you can rerun your code. You can also use the Azure portal to delete indexes, indexers, data sources, and skillsets.

Takeaways

This tutorial demonstrated the basic steps for building an enriched indexing pipeline through the creation of component parts: a data source, skillset, index, and indexer.

[Built-in skills](#) were introduced, along with skillset definition and the mechanics of chaining skills together through inputs and outputs. You also learned that `outputFieldMappings` in the indexer definition is required for routing enriched values from the pipeline into a searchable index on an Azure AI Search service.

Finally, you learned how to test results and reset the system for further iterations. You learned that issuing queries against the index returns the output created by the enriched indexing pipeline. You also learned how to check indexer status, and which objects to delete before rerunning a pipeline.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with all of the objects in an AI enrichment pipeline, let's take a closer look at skillset definitions and individual skills.

[How to create a skillset](#)

Last updated on 11/21/2025

Tutorial: Fix a skillset using Debug Sessions

Article • 12/03/2024

In Azure AI Search, a skillset coordinates the actions of skills that analyze, transform, or create searchable content. Frequently, the output of one skill becomes the input of another. When inputs depend on outputs, mistakes in skillset definitions and field associations can result in missed operations and data.

Debug Sessions is an Azure portal tool that provides a holistic visualization of a skillset that executes on Azure AI Search. Using this tool, you can drill down to specific steps to easily see where an action might be falling down.

In this article, use **Debug Sessions** to find and fix missing inputs and outputs. The tutorial is all-inclusive. It provides sample data, a REST file that creates objects, and instructions for debugging problems in the skillset.

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- Azure AI Search. [Create a service](#) or [find an existing service](#) under your current subscription. You can use a free service for this tutorial. The free tier doesn't provide managed identity support for an Azure AI Search service. You must use keys for connections to Azure Storage.
- Azure Storage account with [Blob storage](#), used for hosting sample data, and for persisting cached data created during a debug session. If you're using a free search service, the storage account must have shared access keys enabled and it must allow public network access.
- [Visual Studio Code](#) with a [REST client](#).
- [Sample PDFs \(clinical trials\)](#).
- [Sample debug-sessions.rest file](#) used to create the enrichment pipeline.

Note

This tutorial also uses [Azure AI services](#) for language detection, entity recognition, and key phrase extraction. Because the workload is so small, Azure AI

services is tapped behind the scenes for free processing for up to 20 transactions. This means that you can complete this exercise without having to create a billable Azure AI services resource.

Set up the sample data

This section creates the sample data set in Azure Blob Storage so that the indexer and skillset have content to work with.

1. Download sample data ([clinical-trials-pdf-19](#)), consisting of 19 files.
2. [Create an Azure Storage account](#) or [find an existing account](#).
 - Choose the same region as Azure AI Search to avoid bandwidth charges.
 - Choose the StorageV2 (general purpose V2) account type.
3. Navigate to the Azure Storage services pages in the Azure portal and create a Blob container. Best practice is to specify the access level "private". Name your container `clinicaltrialdataset`.
4. In container, select **Upload** to upload the sample files you downloaded and unzipped in the first step.
5. While in the Azure portal, copy the connection string for Azure Storage. You can get the connection string from **Settings > Access Keys** in the Azure portal.

Copy a key and URL

This tutorial uses API keys for authentication and authorization. You need the search service endpoint and an API key, which you can get from the Azure portal.

1. Sign in to the [Azure portal](#), navigate to the **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

A valid API key establishes trust, on a per request basis, between the application sending the request and the search service handling it.

Create data source, skillset, index, and indexer

In this section, create a "buggy" workflow that you can fix in this tutorial.

1. Start Visual Studio Code and open the `debug-sessions.rest` file.
2. Provide the following variables: search service URL, search services admin API key, storage connection string, and the name of the blob container storing the PDFs.
3. Send each request in turn. Creating the indexer takes several minutes to complete.
4. Close the file.

Check results in the Azure portal

The sample code intentionally creates a buggy index as a consequence of problems that occurred during skillset execution. The problem is that the index is missing data.

1. In Azure portal, on the search service **Overview** page, select the **Indexes** tab.
2. Select *clinical-trials*.
3. Enter this JSON query string in Search explorer's JSON view. It returns fields for specific documents (identified by the unique `metadata_storage_path` field).

JSON

```
"search": "*",
"select": "metadata_storage_path, organizations, locations",
"count": true
```

- Run the query. You should see empty values for `organizations` and `locations`.

These fields should have been populated through the skillset's [Entity Recognition skill](#), used to detect organizations and locations anywhere within the blob's content. In the next exercise, you'll debug the skillset to determine what went wrong.

Another way to investigate errors and warnings is through the Azure portal.

- Open the **Indexers** tab and select *clinical-trials-idxr*.

Notice that while the indexer job succeeded overall, there were warnings.

- Select **Success** to view the warnings (if there were mostly errors, the detail link would be **Failed**). You'll see a long list of every warning emitted by the indexer.

Status	Last run	Duration	Docs succeeded	Errors/Warnings
✓ Success	1/1/2022, 11:42:52	20.01 s	19/19	0/59

Start your debug session

- From the search service left-navigation pane, under **Search management**, select **Debug sessions**.
- Select **+ Add Debug Session**.
- Give the session a name.
- In Indexer template, provide the indexer name. The indexer has references to the data source, the skillset, and index.
- Select the storage account.
- Save the session.

Create debug session

X

Debug session name

Description

Indexer template



Specify a blob storage location where the debug session can write output artifacts. Prior debug session data may be overwritten.

Subscription



Storage account



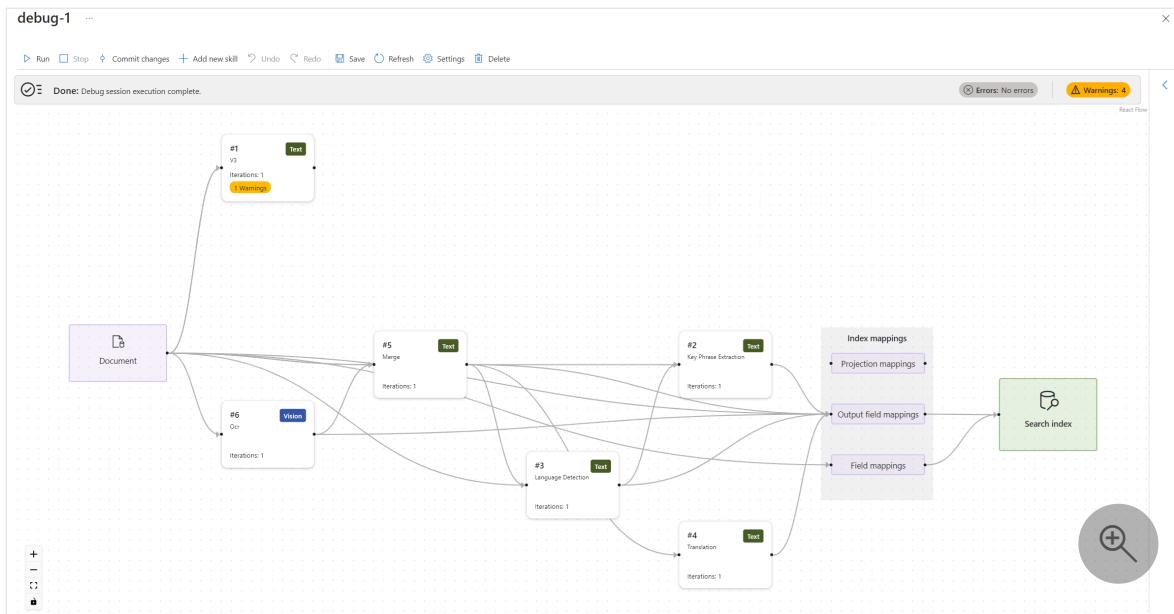
Authenticate using managed identity. [Learn more](#)

Save

Cancel



7. A debug session opens to the settings page. You can make modifications to the initial configuration and override any defaults. A debug session only works with a single document. The default is to accept the first document in the collection as the basis of your debug sessions. You can [choose a specific document to debug](#) by providing its URI in Azure Storage.
8. When the debug session has finished initializing, you should see a skills workflow with mappings and a search index. The enriched document data structure appears in a details pane on the side. We excluded it from the following screenshot so that you could see more of the workflow.



Find issues with the skillset

Any issues reported by the indexer are indicated as **Errors** and **Warnings**.

Notice that the number of errors and warning is a much smaller list than the one displayed earlier because this list is only detailing the errors for a single document. Like the list displayed by the indexer, you can select on a warning message and see the details of this warning.

Select **Warnings** to review the notifications. You should see four:

- "Could not execute skill because one or more skill inputs were invalid. Required skill input is missing. Name: 'text', Source: '/document/content'."
- "Could not map output field 'locations' to search index. Check the 'outputFieldMappings' property of your indexer. Missing value '/document/merged_content/locations'."
- "Could not map output field 'organizations' to search index. Check the 'outputFieldMappings' property of your indexer. Missing value '/document/merged_content/organizations'."
- "Skill executed but may have unexpected results because one or more skill inputs were invalid. Optional skill input is missing. Name: 'languageCode', Source: '/document/languageCode'. Expression language parsing issues: Missing value '/document/languageCode'."

Many skills have a "languageCode" parameter. By inspecting the operation, you can see that this language code input is missing from the `EntityRecognitionSkill.#1`, which is

the same entity recognition skill that is having trouble with 'locations' and 'organizations' output.

Because all four notifications are about this skill, your next step is to debug this skill. If possible, start by solving input issues first before moving on to output issues.

Fix missing skill input values

1. On the work surface, select the skill that's reporting the warnings. In this tutorial, it's the entity recognition skill.
2. The Skill details pane opens to the right with sections for iterations and their respective inputs and outputs, skill settings for the JSON definition of the skill, and messages for any errors and warnings that this skill is emitting.

The screenshot shows the 'Skill: #1' details pane. At the top, there are three tabs: 'Iterations (1)', 'Skill Settings', and 'Errors/Warnings (1)'. The 'Iterations (1)' tab is selected and underlined. Below the tabs is a table with the following data:

Iteration	Property	Name	Path
0	Inputs	text	=\${/document/content}
		languageCode	=\${/document/languageCode}
	Outputs	organizations	/document/organizations
		locations	/document/locations

At the bottom of the pane are two buttons: 'Save' (blue) and 'Cancel'.

3. Hover over each input (or select an input) to show the values in the **Expression evaluator**. Notice that the displayed result for this input doesn't look like a text input. It looks like a series of new line characters `\n \n\n\n\n` instead of text. The lack of text means that no entities can be identified, so either this document fails to meet the prerequisites of the skill, or there's another input that should be used instead.

The screenshot shows the 'Expression evaluator' interface. On the left, under 'Value', there is a red warning icon followed by the text "\n \n\n\n\n\n". Below this, under 'Path', several paths are listed: \$(/document/content), \$(/document/languageCode), /document/organizations, and /document/locations.

- Switch back to **Enriched data structure** and review the enrichment nodes for this document. Notice the `\n \n\n\n\n` for "content" has no originating source, but another value for "merged_content" has OCR output. Although there's no indication, the content of this PDF appears to be a JPEG file, as evidenced by the extracted and processed text in "merged_content".

Enriched data structure

content	"\n \n\n\n\n\n"
language	"en"
locations	[]
➤ merged_content	"\n \n\n Study of BMN 110 in Pediatric Patients < 5 Years of Age With Mucopolysaccharidosis IVA (Morquio..."
metadata_author	"Azure Search"
metadata_content_type	"application/pdf"
metadata_creation_date	"2019-09-04T14:24:13Z"
metadata_language	"cy"
metadata_storage_content_md5	"/TujjHwdgZRV+G6k4TSxdw=="
metadata_storage_content_type	"application/pdf"
metadata_storage_file_extension	".pdf"
metadata_storage_last_modified	"2024-04-19T22:11:32Z"
metadata_storage_name	"ICT01515956.pdf"
metadata_storage_path	"aHR0cHM6Ly9oZWlkaXN0b3JhZ2VkZW1vLmJsb2luY29yZS?sv=2019-07-07&sr=b&sig=yzYt6jU47Ie%2F0ImavixERtl2nkWeysMV3Jb08-21T18%3A10%3A4..."
metadata_storage_size	76451
metadata_title	"Study of BMN 110 in Pediatric Patients < 5 Years of Age With"

- Switch back to the skill and select **Skillset settings** to open the JSON definition.
- Change the expression from `/document/content` to `/document/merged_content`, and then select **Save**. Notice that the warning is no longer listed.

```

Skill: #1
Iterations (1) Skill Settings Errors/Warnings (1)

1 {
2   "odataType": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
3   "name": "#1",
4   "context": "/document",
5   "inputs": [
6     {
7       "name": "text",
8       "source": "/document/content",
9       "inputs": []
10    },

```



```

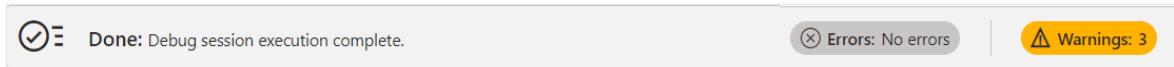
Skill: #1
Iterations (1) Skill Settings Errors/Warnings (1)

1 {
2   "odataType": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
3   "name": "#1",
4   "context": "/document",
5   "inputs": [
6     {
7       "name": "text",
8       "source": "/document/merged_content",
9       "inputs": []
10    },

```

7. Select **Run** in the session's window menu. This kicks off another execution of the skillset using the document.

8. Once the debug session execution completes, notice that the warnings count has reduced by one. Warnings show that the error for text input is gone, but the other warnings remain. The next step is to address the warning about the missing or empty value `/document/languageCode`.



9. Select the skill and hover over `/document/languageCode`. The value for this input is null, which isn't a valid input.

10. As with the previous issue, start by reviewing the **Enriched data structure** for evidence of its nodes. Notice that there's no "languageCode" node, but there's one for "language". So, there's a typo in the skill settings.

11. Copy the expression `/document/language`.

12. In the Skill details pane, select **Skill Settings** for the #1 skill and paste the new value, `/document/language`.

13. Select **Save**.

14. Select **Run**.

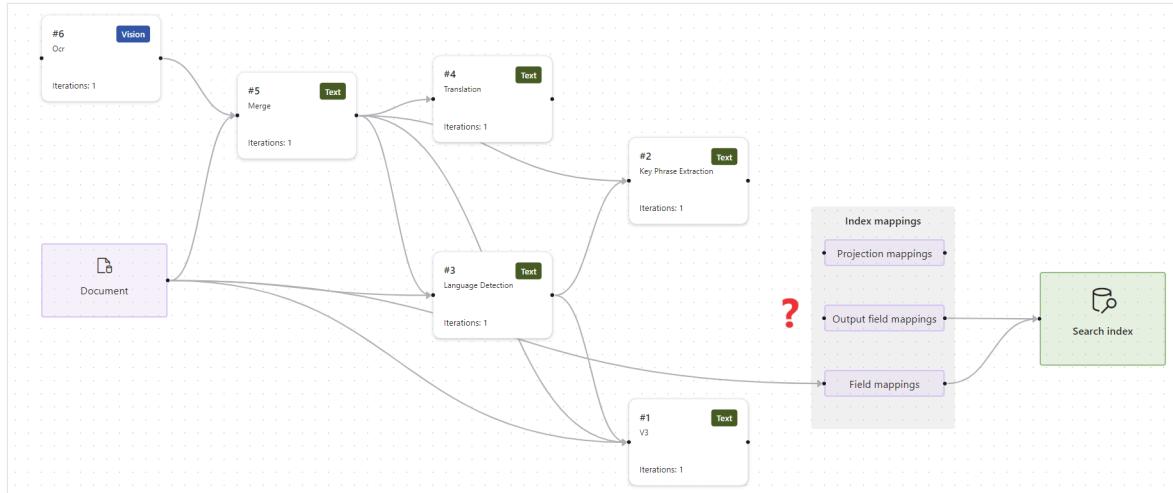
15. After the debug session execution completes, you can check the results in the Skills detail pane. When you hover over `/document/language`, you should see `en` as the value in the **Expression evaluator**.

Notice that the input warnings are gone. There now remain just the two warnings about output fields for organizations and locations.

Fix missing skill output values

The messages say to check the 'outputFieldMappings' property of your indexer, so lets start there.

1. Select **Output Field Mappings** on the work surface. Notice that the output field mappings are missing.



2. As a first step, confirm that the search index has the expected fields. In this case, the index has fields for "locations" and "organizations".
3. If there's no problem with the index, the next step is to check skill outputs. As before, select the **Enriched data structure**, and scroll the nodes to find "locations" and "organizations". Notice that the parent is "content" instead of "merged_content". The context is wrong.
4. Switch back to Skills detail pane for the entity recognition skill.
5. In **Skill Settings**, change `context` to `document/merged_content`. At this point, you should have three modifications to the skill definition altogether.

Skill: #1

X

Iterations (1) Skill Settings Errors/Warnings (0)

```
1  {
2      "odataType": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
3      "name": "#1",
4      "context": "/document/merged_content",
5      "inputs": [
6          {
7              "name": "text",
8              "source": "/document/merged_content",
9              "inputs": []
10         },
11         {
12             "name": "languageCode",
13             "source": "/document/language",
14             "inputs": []
15         }
16     ],
17     "outputs": [
18         {
19             "name": "organizations",
20             "targetName": "organizations"
21         },
22         {
23             "name": "locations",
24             "targetName": "locations"
25         }
26     ],
27     "categories": [
28         "Organization",
29         "Location"
30     ],
31     "defaultLanguageCode": "en"
32 }
```

Save

Cancel

6. Select Save.

7. Select Run.

All of the errors have been resolved.

Commit changes to the skillset

When the debug session was initiated, the search service created a copy of the skillset. This was done to protect the original skillset on your search service. Now that you have

finished debugging your skillset, the fixes can be committed (overwrite the original skillset).

Alternatively, if you aren't ready to commit changes, you can save the debug session and reopen it later.

1. Select **Commit changes** in the main Debug sessions menu.
2. Select **OK** to confirm that you wish to update your skillset.
3. Close Debug session and open **Indexers** from the left pane.
4. Select 'clinical-trials-idxr'.
5. Select **Reset**.
6. Select **Run**.
7. Select **Refresh** to show the status of the reset and run commands.

When the indexer has finished running, there should be a green checkmark and the word Success next to the time stamp for the latest run in the **Execution history** tab. To ensure that the changes have been applied:

1. In the left pane, open **Indexes**.
2. Select 'clinical-trials' index and in the Search explorer tab, enter this query string:
`$select=metadata_storage_path, organizations, locations&$count=true` to return fields for specific documents (identified by the unique `metadata_storage_path` field).
3. Select **Search**.

The results should show that organizations and locations are now populated with the expected values.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

The free service is limited to three indexes, indexers, and data sources. You can delete individual items in the Azure portal to stay under the limit.

Next steps

This tutorial touched on various aspects of skillset definition and processing. To learn more about concepts and workflows, refer to the following articles:

- [How to map skillset output fields to fields in a search index](#)
 - [Skillsets in Azure AI Search](#)
 - [How to configure caching for incremental enrichment](#)
-

Feedback

Was this page helpful?



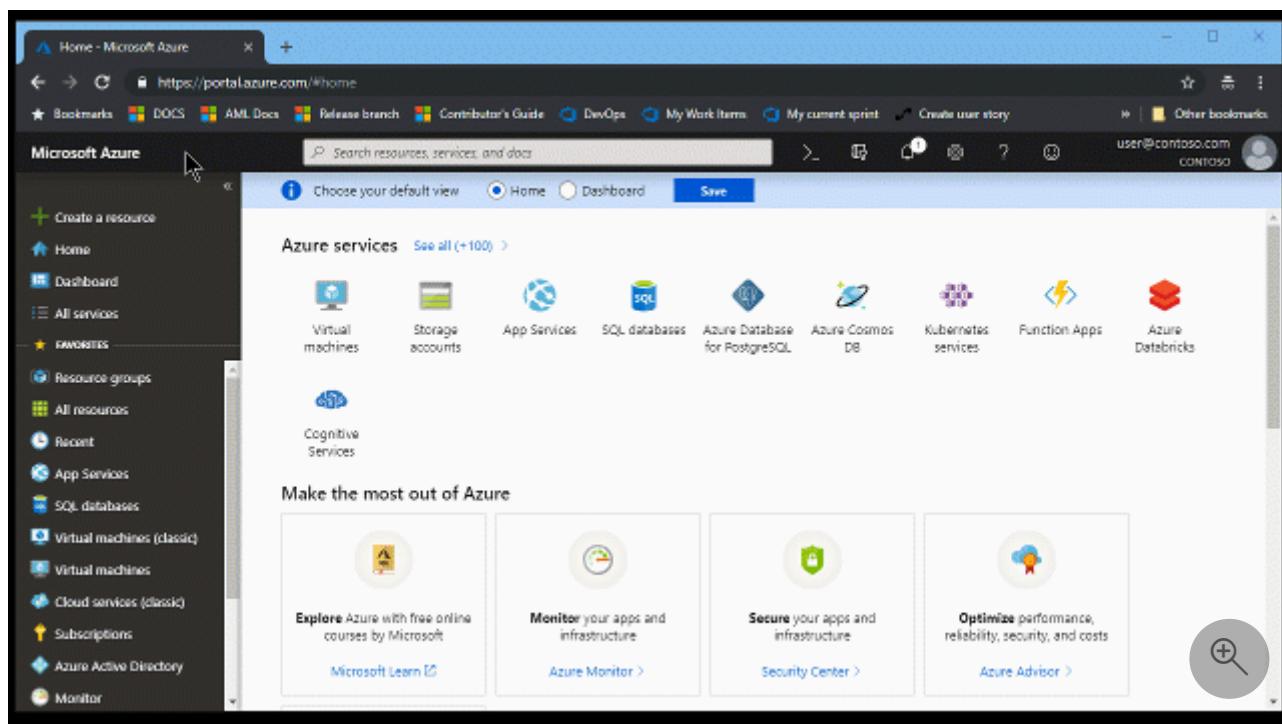
[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create an Azure AI Search service in the Azure portal

09/30/2025

Azure AI Search is an information retrieval platform for the enterprise. It supports traditional search and conversational, AI-driven search for "chat with your data" experiences across your proprietary content.

The easiest way to create a search service is through the [Azure portal](#), which is covered in this article.



You can also use:

- [Azure PowerShell](#)
- [Azure CLI](#)
- [Management REST API](#)
- [Azure Resource Manager template](#)
- [Bicep](#)
- [Terraform](#)

Before you start

Some properties are fixed for the lifetime of the search service. Before you create your service, decide on the following properties:

Property	Description
Name	Becomes part of the URL endpoint. The name must be unique and follow naming rules.
Region	Determines data residency and availability of certain features. For example, semantic ranker and Azure AI integration have region requirements. Choose a region that supports the features you need.
Tier	Determines infrastructure, service limits, and billing. Some features aren't available on lower or specialized tiers. After you create your service, you can switch between Basic and Standard (S1, S2, and S3) tiers .
Compute type	Determines virtualization and security model. You can choose between standard VMs (recommended) and confidential VMs, which are intended for select workloads requiring data-in-use privacy and isolation.

Subscribe to Azure

Azure AI Search requires a free or Standard Azure subscription.

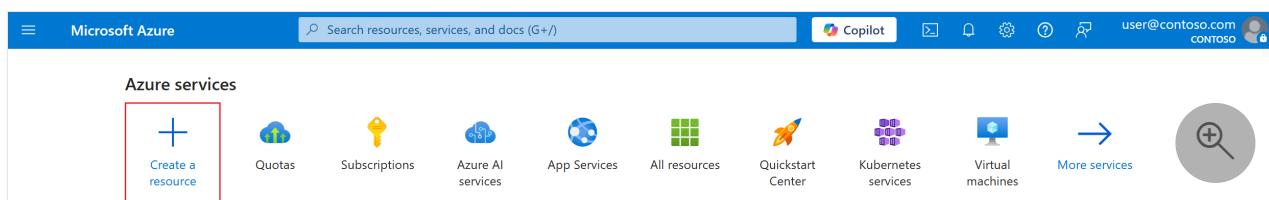
To try Azure AI Search for free, [start a trial subscription](#) and then [create your search service on the Free tier](#). Each Azure subscription can have one free search service, which is intended for short-term, non-production evaluation of the product. You can complete all of our quickstarts and most of our tutorials on the Free tier. For more information, see [Try Azure AI Search for free](#).

ⓘ Important

To make room for other services, Microsoft might delete free services that are inactive for an extended period of time.

Find the Azure AI Search offering

1. Sign in to the [Azure portal](#).
2. In the upper-left corner of your dashboard, select **Create a resource**.



3. Use the search box to find Azure AI Search.

Home > Marketplace ...

Get Started

Service Providers

Management

Private Marketplace

Private Offer Management

My Marketplace

Favorites

My solutions

Recently created

Private plans

Categories

AI + Machine Learning (137)

Azure AI Search

Pricing : All

Operating System : All

Publisher Type : All

Product Type : All

Publisher name : All

Showing 1 to 20 of 207 results for 'Azure AI Search'. [Clear search](#)

Azure AI Search

Microsoft

Azure Service

AI-powered cloud search service for mobile and web app development (formerly Azure Cognitive Search)

Create

AI Search

Aisera

SaaS

LLM-powered search for personalized, accurate, & permission-aware results across all data sources

Price varies

Subscribe

Elastic Search (Elasticsearch) – An Azure Native ISV

Elastic

SaaS

The Elastic Search AI Platform brings together the precision of search and the intelligence of AI.

Price varies

Subscribe

BA Insight for Azure AI Search

BA-Insight - Global HQ (Boston)

SaaS

Creating relevant, personalized, actionable, and intelligent AI content search is easier than ever.

Price varies

Subscribe

Evoke Workspace Search - EWS

Evoke Technologies Private Limit...

Azure Application

EWS is an AI-powered platform that uses GenAI combining large language models with enterprise data.

Starts at \$249.00/month

Create

Choose a subscription

If you have multiple Azure subscriptions, choose one for your search service.

If you're implementing [customer-managed encryption](#) or using other features that rely on managed service identities for [external data access](#), choose the same subscription you use for Azure Key Vault or other services that use managed identities.

Set a resource group

A resource group is a container that holds related resources for an Azure solution. Use it to consolidate same-solution resources, monitor costs, and check the creation date of your search service.

Create a search service

Basics Scale Networking Tags Review + create

Project details

Subscription * <your-subscription-name-appears-here>

Resource Group *

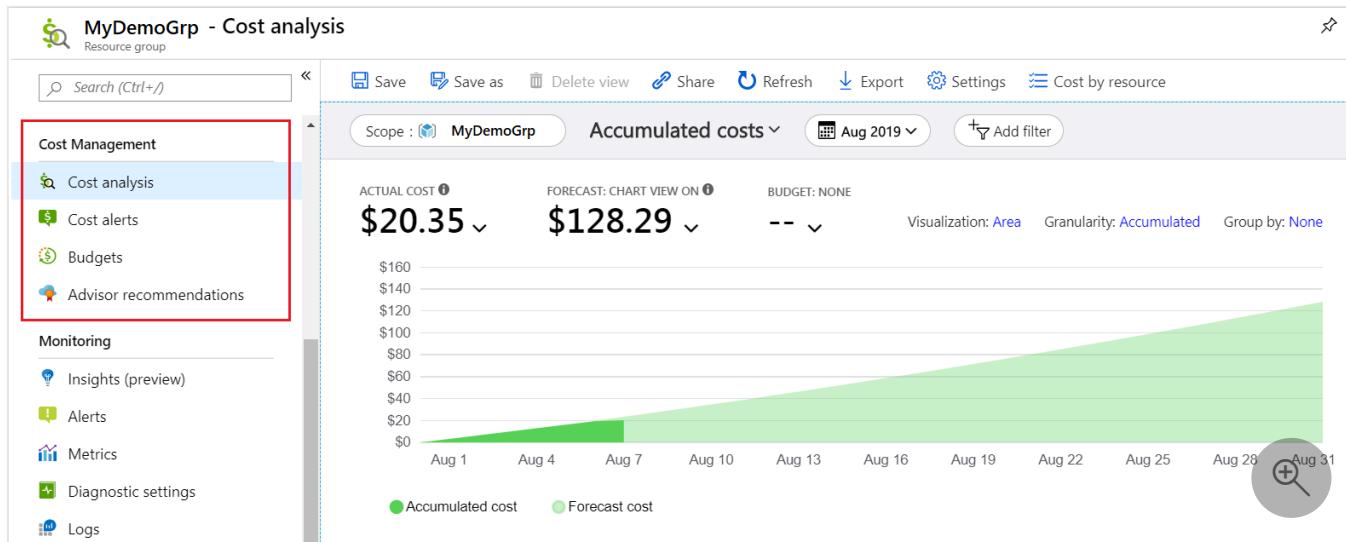
Create new

A resource group is a container that holds related resources for an Azure solution.

Name * my-resource-group

OK Cancel

Over time, you can track current and projected costs for individual resources and for the overall resource group. The following screenshot shows the cost information that's available when you combine multiple resources into one group:



Name your service

Enter a name for your search service. The name is part of the endpoint against which API calls are issued: <https://your-service-name.search.windows.net>. For example, if you enter `myservice`, the endpoint becomes <https://myservice.search.windows.net>.

When naming your service, follow these rules:

- Use a name that's unique within the `search.windows.net` namespace.
- Use between 2 and 60 characters.
- Use only lowercase letters, digits, and dashes (-).
- Don't use dashes as the first two characters or the last character.
- Don't use consecutive dashes.

Tip

If you have multiple search services, it's helpful to include the region in the service name. For example, when deciding how to combine or attach resources, the name `myservice-westus` might save you a trip to the Properties page.

Choose a region

Important

Due to high demand, Azure AI Search is currently unavailable for new instances in some regions.

If you use multiple Azure services, putting all of them in the same region minimizes or voids bandwidth charges. There are no charges for data egress among same-region services.

In most cases, choose a region near you, unless any of the following apply:

- Your nearest region is [at capacity](#), which is indicated by the footnotes of each table. The Azure portal has the advantage of hiding unavailable regions and tiers during resource setup.
- You want to use integrated data chunking and vectorization or built-in skills for AI enrichment. Integrated operations have region requirements.
- You want to use Azure Storage for indexer-based indexing, or you want to store application data that isn't in an index. Debug session state, enrichment caches, and knowledge stores are Azure AI Search features that depend on Azure Storage. The region you choose for Azure Storage has implications for network security. If you're setting up a firewall, you should place the resources in separate regions. For more information, see [Outbound connections from Azure AI Search to Azure Storage](#).

Checklist for choosing a region

1. Is Azure AI Search available in a nearby region? Check the [list of supported regions](#).
2. Do you have a specific tier in mind? Check [region availability by tier](#).
3. Do you have business continuity and disaster recovery (BCDR) requirements? Create two or more search services in different Azure regions, each with two or more replicas so that they can be spread across multiple [availability zones](#). For example, if you're operating in North America, you might choose East US and West US, or North Central US and South Central US, for each search service. For more information, see [Multi-region deployments in Azure AI Search](#).
4. Do you need [AI enrichment](#), [integrated data chunking and vectorization](#), or [multimodal search](#)? For [billing purposes](#), Azure AI Search and Azure AI services multi-service must coexist in the same region.
 - Check [Azure AI Search regions](#). If you're using OCR, entity recognition, or other skills backed by Azure AI, the **AI enrichment** column indicates whether Azure AI Search and Azure AI services multi-service are in the same region.
 - Check [Azure AI Vision regions](#) for multimodal APIs that enable text and image vectorization. These APIs are powered by Azure AI Vision and accessed through an Azure AI services multi-service resource. However, they're generally available in fewer regions than the multi-service resource itself.

Choose a tier

Azure AI Search is offered in multiple [pricing tiers](#):

- Free
- Basic
- Standard
- Storage Optimized

Each tier has its own [capacity and limits](#), and some features are tier dependent. For information about computing characteristics, feature availability, and region availability, see [Choose a service tier for Azure AI Search](#).

The Basic and Standard tiers are the most common for production workloads, but many customers start with the Free tier. The billable tiers differ primarily in partition size, partition speed, and limits on the number of objects you can create.

Select Pricing Tier

Browse available skus and their features

Sku	Offering	Indexes	Indexers	Vector quota	Total storage	Search units	Replicas	Partitions
F	Free	3	3	25 MB 	50 MB	1	1	1
B	Basic	15	15	5 GB/Partition	15 GB/Partition	9	3	3
S	Standard	50	50	35 GB/Partition	160 GB/Partition	36	12	12
S2	Standard	200	200	150 GB/Partition	512 GB/Partition	36	12	12
S3	Standard	200	200	300 GB/Partition	1 TB/Partition	36	12	12
S3HD	High-density	1000	0	300 GB/Partition	1 TB/Partition	36	12	3
L1	Storage Optimized	10	10	150 GB/Partition	2 TB/Partition	36	12	12
L2	Storage Optimized	10	10	300 GB/Partition	4 TB/Partition	36	12	12

 Higher storage limits are available for new services in this region at no additional cost.



Note

Services created after April 3, 2024 have larger partitions and higher vector quotas at every billable tier.

Choose a compute type

The compute type determines the virtualization and security model used to deploy your search service. There are two compute types:

- **Default** (base cost) deploys your search service on standard Azure infrastructure, encrypting data at rest and in transit but not in use. Recommended for most search workloads.
- **Confidential** (10% surcharge) uses [Azure confidential computing](#) to isolate processing in a hardware-based trusted execution environment, protecting unencrypted data in use from unauthorized access. Recommended only if you have advanced privacy, compliance, or regulatory requirements.

Confidential computing has limited regional availability, disables or restricts certain features, and increases the cost of running your search service. For a detailed comparison of both compute types, see [Data in use](#).

Create your service

After providing the necessary inputs, create your search service.

Create a search service

...

[Basics](#) [Scale](#) [Networking](#) [Tags](#) [Review + create](#)

Project details

Subscription *

<your-subscription-name-appears-here>



Resource Group *

my-resource-group

[Create new](#)

Instance Details

Service name * ⓘ

my-service-name



Location *

Central US



Pricing tier * ⓘ

Standard

160 GB/Partition, max 12 replicas, max 12 partitions, max 36 search units

[Change Pricing Tier](#)[Review + create](#)[Previous](#)[Next: Scale](#)

Your service is deployed within minutes, and you can monitor its progress with Azure notifications. Consider pinning the service to your dashboard for easy access in the future.

The screenshot shows the Azure Notifications panel. At the top, there are icons for a file, a bell (highlighted with a red box), a gear, a question mark, and a user profile. To the right of the icons, it displays the email address user@contoso.com and the name **CONTOSO**. Below the header, the title "Notifications" is displayed next to a close button ("X").

Below the title, there is a message: "More events in the activity log →" followed by a "Dismiss all" button with a dropdown arrow.

A single notification is listed: "Deployment succeeded" (indicated by a green checkmark icon). The message states: "Deployment 'my-search-service' to resource group 'my-resource-group' was successful."

At the bottom of the notification, there are two buttons: "Go to resource" (highlighted with a red box) and "Pin to dashboard". A timestamp "29 minutes ago" is shown at the bottom right, along with a circular search icon.

Configure authentication

When you create a search service, key-based authentication is the default, but it's not the most secure option. We recommend that you replace it with role-based access.

To enable role-based access for your service:

1. Go to your search service in the [Azure portal](#).
2. From the left pane, select **Settings > Keys**. You can connect to your service using [API keys](#), [Azure roles](#), or both. Select **Both** until you assign roles, after which you can select **Role-based access control**.

The screenshot shows the Azure portal interface for managing a search service named 'my-search-service'. The left sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Search management, Settings, Semantic ranker, Knowledge Center, and Keys. The 'Keys' option is currently selected and highlighted with a red box. On the right, under the 'Keys' section, there's a 'Manage admin keys' area with fields for Primary admin key and Secondary admin key, each with a 'Regenerate' button. Below this is a 'Manage query keys' area with 'Add' and 'Delete' buttons, and a table showing columns for Name and Key. A circular icon with a magnifying glass and a download arrow is located at the bottom right of the table.

Scale your service

After deploying your search service, you can [scale it to meet your needs](#). Azure AI Search offers two scaling dimensions: *replicas* and *partitions*. Replicas allow your service to handle a higher load of search queries, while partitions allow your service to store and search through more documents.

Scaling is available only on billable tiers. On the Free tier, you can't scale your service or configure replicas and partitions.

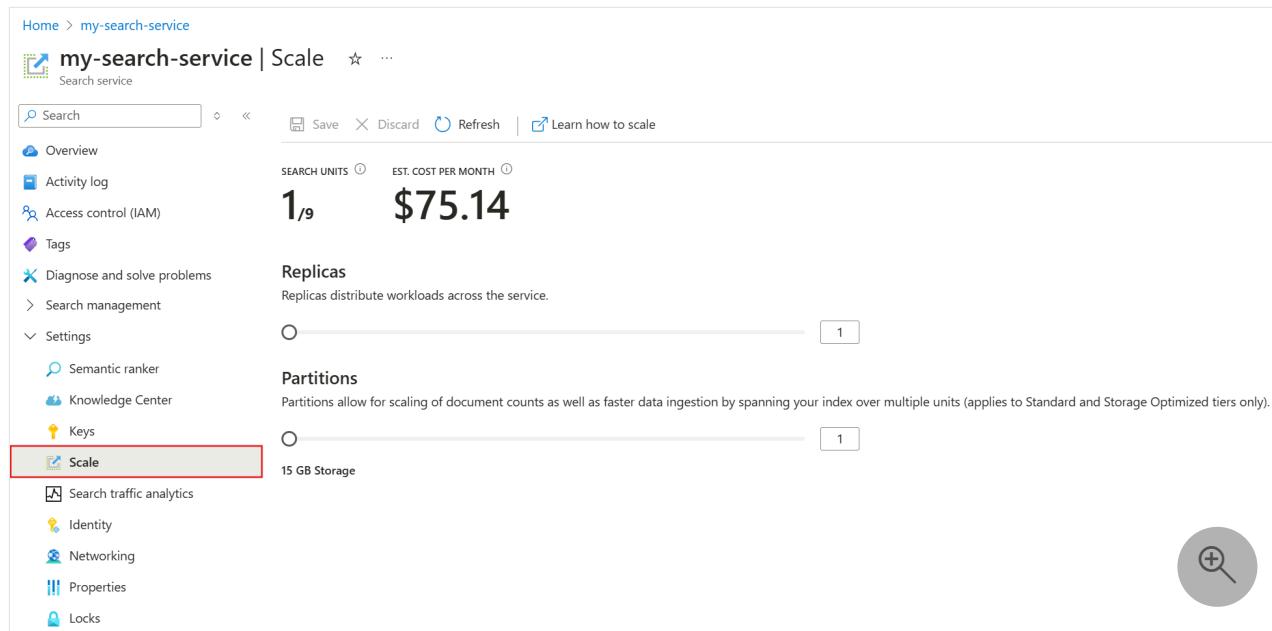
i Important

Your service must have [two replicas for read-only SLA](#) and [three replicas for read/write SLA](#).

Adding resources will increase your monthly bill. Use the [pricing calculator](#) to understand the billing implications. You can adjust resources based on load, such as increasing resources for initial indexing and decreasing them later for incremental indexing.

To scale your service:

1. Go to your search service in the [Azure portal](#).
2. From the left pane, select **Settings > Scale**.



The screenshot shows the Azure portal interface for scaling a search service. At the top, it says "Home > my-search-service". Below that is the title "my-search-service | Scale". On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Search management, Semantic ranker, Knowledge Center, Keys, and Scale. The "Scale" option is highlighted with a red box. The main area shows "SEARCH UNITS" at 1,9 and "EST. COST PER MONTH" at \$75.14. There are two sliders: one for "Replicas" set to 1, and another for "Partitions" also set to 1. A note below the partitions slider says "Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple units (applies to Standard and Storage Optimized tiers only)". At the bottom right is a magnifying glass icon.

3. Use the sliders to add replicas and partitions.

When to add a second service

Most customers use a single search service at a tier [sufficient for the expected load](#). One service can host multiple indexes, each isolated from the others, within the [maximum limits of your chosen tier](#). In Azure AI Search, you can direct requests to only one index, reducing the chance of retrieving data from other indexes in the same service.

However, you might need a second service for the following operational requirements:

- Region outages. In the unlikely event of a full region outage, Azure AI Search doesn't provide instant failover. You must implement your own multi-region solution and failover approach. For more information, see [Multi-region deployments in Azure AI Search](#).
- [Multitenant architectures](#) that require two or more services.

- Globally deployed applications that require services in each geography to minimize latency.

! Note

In Azure AI Search, you can't separate indexing and querying operations, so don't create multiple services for separate workloads. An index is always queried on the service in which it was created, and you can't copy an index to another service.

A second service isn't required for high availability. You achieve high availability for queries by using two or more replicas in the same service. Because the replicas are updated sequentially, at least one is operational when a service update is rolled out. For more information about uptime, see [Service Level Agreements](#).

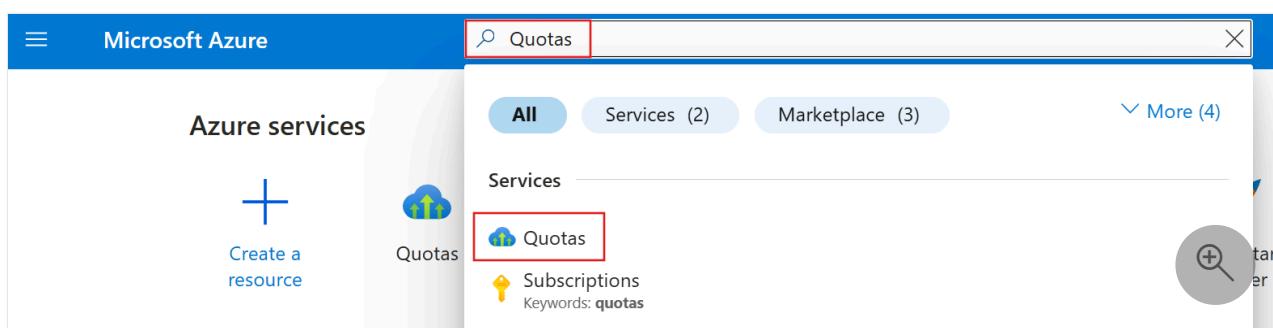
Add more services to your subscription

Azure AI Search limits the [number of search services](#) you can initially create in a subscription. If you reach your limit, you can request more quotas.

You must have Owner or Contributor permissions for the subscription to request quota. Depending on your region and data center capacity, you might be able to automatically request quota to add services to your subscription. If the request fails, reduce the number or file a support ticket. Expect a one-month turnaround for a large quota increase, such as more than 30 extra services.

To request more subscription quota:

1. Go to your dashboard in the [Azure portal](#).
2. Use the search box to find the **Quotas** service.



3. On the **Overview** tab, select the **Search** tile.

The screenshot shows the 'Quotas' section of the Azure portal. On the left, there's a sidebar with 'Settings' and links for 'My quotas', 'Fired alerts (Preview)', and 'Alert rules (Preview)'. The main area has tabs for 'Overview' and 'Requests'. A title 'View and adjust my quotas' is followed by a note: 'Don't see your products below? Go to Azure subscription and service limits, quotas, and constraints' and 'and check the list in the right pane for your products. For further assistance, contact the support team.' Below this are several service icons in a grid: Compute, Compute (classic), Networking, Machine learning, Storage, Storage (classic), HPC Cache, Azure HDInsight, Azure Lab Services, Microsoft Purview, Azure Container Instances, Dev Box, Azure Container Apps, App Services, and a 'Search' button which is highlighted with a red box.

- Set filters to review the existing quota for search services in your current subscription. We recommend filtering by usage.

The screenshot shows the 'Quotas | My quotas' page. The sidebar includes 'Overview', 'Settings', and 'My quotas' (which is selected). The main table shows quota details for 'Usage at or near quota' and 'Usage at low level'. A modal window titled 'Usage : Only show items with usage' is open, containing three radio button options: 'Show all' (unchecked), 'Only show items with usage' (checked), and 'Select custom usage' (unchecked). The 'Apply' button is highlighted with a red box. To the right of the modal, there's a 'Request adjustment' section with a search icon.

- Next to the tier and region that need more quotas, select Request adjustment.
- In New Quota Request, enter a new limit for your subscription quota. The new limit must be greater than your current limit. If regional capacity is constrained, your request won't be automatically approved, and an incident report will be generated on your behalf for investigation and resolution.
- Submit your request.
- Monitor notifications in the Azure portal for updates on the new limit. Most requests are approved within 24 hours.

Next steps

Now that you've deployed your search service, continue in the Azure portal to create your first index:

[Quickstart: Create an Azure AI Search index in the Azure portal](#)

Want to optimize and save on your cloud spending?

[Start analyzing costs with Cost Management](#)

Configure your Azure AI Search service in the Azure portal

Configuring your new Azure AI Search service involves several tasks to optimize security, access, and performance. This article provides a day-one checklist to help you set up your service in the [Azure portal](#).

After you create a search service, we recommend that you:

- ✓ [Configure role-based access](#)
- ✓ [Configure a managed identity](#)
- ✓ [Configure network security](#)
- ✓ [Check capacity and understand billing](#)
- ✓ [Enable diagnostic logging](#)
- ✓ [Provide connection information to developers](#)

Configure role-based access

Portal access is based on [role assignments](#). By default, new search services have at least one service administrator or owner. Service administrators, co-administrators, and owners have permission to create more administrators and assign other roles. They also have access to all portal pages and operations on default search services.

Tip

By default, any administrator or owner can create or delete services. To prevent accidental deletions, consider [locking your resources](#).

Each search service comes with [API keys](#) and uses key-based authentication by default. However, we recommend using Microsoft Entra ID and role-based access control (RBAC) for improved security. RBAC eliminates the need to store and pass API keys in plain text.

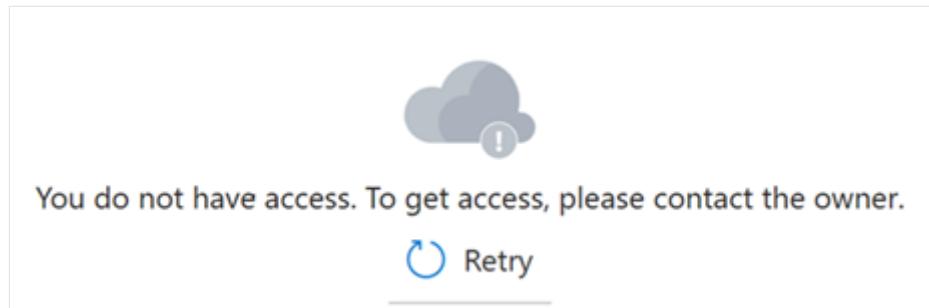
When you switch from key-based authentication to keyless authentication, service administrators must assign themselves data plane roles for full access to objects and data. These roles include Search Service Contributor, Search Index Data Contributor, and Search Index Data Reader.

To configure role-based access:

1. [Enable roles](#) on your search service. We recommend using both API keys and roles.

2. [Assign data plane roles](#) to replace the functionality lost when you disable API keys. An owner only needs Search Index Data Reader, but developers need [more roles](#).

Role assignments can take several minutes to take effect. Until then, portal pages used for data plane operations display the following message:



3. [Assign more roles](#) for solution developers and apps.

Configure a managed identity

If you plan to use indexers for automated indexing, applied AI, or integrated vectorization, you should [configure your search service to use a managed identity](#). You can then assign roles on other Azure services that authorize your search service to access data and operations.

For integrated vectorization, your search service identity needs the following roles:

- Storage Blob Data Reader on Azure Storage
- Cognitive Services Data User on a Microsoft Foundry resource

Role assignments can take several minutes to take effect.

Before you move on to network security, consider testing all points of connection to validate role assignments. Run an [import wizard](#) to test permissions.

Configure network security

By default, a search service accepts authenticated and authorized requests over public internet connections. You have two options for enhancing network security:

- [Configure firewall rules](#) to restrict network access by IP address.
- [Configure a private endpoint](#) to only allow traffic from Azure virtual networks. Note that when you turn off the public endpoint, the import wizards won't run.

To learn about inbound and outbound calls in Azure AI Search, see [Security in Azure AI Search](#).

Check capacity and understand billing

By default, a search service is created with one replica and one partition. You can [add capacity](#) by adding replicas and partitions, but we recommend waiting until volumes require it. Many customers run production workloads on the minimum configuration.

Semantic ranker can increase the cost of running your service if you opt into the standard plan. If you don't want to use this feature, you can [disable semantic ranker](#) at the service level.

To learn about other features that affect billing, see [How you're charged for Azure AI Search](#).

Enable diagnostic logging

[Enable diagnostic logging](#) to track user activity. If you skip this step, you still get [activity logs](#) and [platform metrics](#) automatically. However, if you want index and query usage information, you should enable diagnostic logging and choose a destination for logged operations. We recommend Log Analytics Workspace for durable storage so that you can run system queries in the Azure portal.

Internally, Microsoft collects telemetry data about your service and the platform. To learn more about data retention, see [Retention of metrics](#).

To learn more about data location and privacy, see [Data residency](#).

Enable semantic ranker

Semantic ranker is free for the first 1,000 requests per month. It's enabled by default on newer search services.

To enable semantic ranker in the portal, select **Settings > Premium features** from the left pane, and then select the **Free** plan. For more information, see [Enable semantic ranker](#).

Provide connection information to developers

To connect to Azure AI Search, developers need:

- An endpoint or URL from the [Overview](#) page.
- An API key from the [Keys](#) page or a role assignment. We recommend Search Service Contributor, Search Index Data Contributor, and Search Index Data Reader.

We recommend portal access for the [import wizards](#) and [Search explorer](#). You must be a contributor or higher to run the wizards.

Related content

For programmatic support for service administration, see the following APIs and modules:

- [Management REST API reference](#)
- [Az.Search PowerShell module](#)
- [az search Azure CLI module](#)

You can also use the management client libraries in the Azure SDKs for .NET, Python, Java, and JavaScript.

There's feature parity across all modalities and languages, except for preview management features. As a general rule, preview management features are released through the Management REST API first.

Last updated on 11/18/2025

Azure AI Search regions list

This article identifies the cloud regions in which Azure AI Search is available. It also lists which premium features are available in each region.

Features subject to regional availability

When you create an Azure AI Search service, your region selection might depend on features that are only available in certain regions. The following table lists those region-specific features.

 Expand table

Feature	Description	Availability
AI enrichment	Refers to built-in skills that make internal calls to Foundry Tools for enrichment and transformation during indexing. Integration requires that Azure AI Search coexists with a Microsoft Foundry resource in the same physical region. You can bypass region requirements by using identity-based connections , currently in public preview.	Regional support is noted in this article.
Availability zones	Divides a region's data centers into distinct physical location groups, providing high availability within the same geo.	Regional support is noted in this article.
Agentic retrieval	Uses the agentic retrieval engine designed for conversational search.	Regional support is noted in this article.
Confidential computing	Deploys your search service on confidential VMs to process data in a hardware-based trusted execution environment. Confidential computing disables or restricts certain features, including agentic retrieval, semantic ranker, query rewrite, and skillset execution.	Regional support is noted in this article.
Semantic ranker	Takes a dependency on Microsoft-hosted models in specific regions.	Regional support is noted in this article.
Query rewrite	Takes a dependency on Microsoft-hosted models in specific regions.	Regional support is noted in this article.
Extra capacity	Higher-capacity partitions became available in select regions starting in April 2024, with a second wave following in May 2024. Currently, there are just a few regions that <i>don't</i> offer higher-capacity partitions. If you have an older search service in a supported region, check if you can upgrade your service . Otherwise, create a	Regional support is noted in the footnotes of this article.

Feature	Description	Availability
	new search service to benefit from more capacity at the same billing rate.	
Capacity constraints	In some regions, insufficient capacity prevents you from creating search services on certain tiers. The Azure portal automatically hides regions and tiers that aren't available for new deployments.	Regional support is noted in the footnotes of this article.
Azure Vision in Foundry Tools 4.0 multimodal APIs	Refers to the Azure Vision multimodal embeddings skill and vectorizer that call the multimodal embedding API.	Check the Azure Vision region list first, and then verify Azure AI Search is available in the same region.

Azure Public regions

You can create an Azure AI Search service in any of the following Azure public regions. Almost all of these regions support [higher-capacity tiers](#). Exceptions are noted where applicable.

Americas

[] Expand table

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Brazil South ¹	✓		✓	✓	✓	✓
Canada Central ¹	✓	✓	✓	✓	✓	✓
Canada East ¹			✓		✓	
Central US	✓	✓	✓		✓	✓
East US ^{1, 2}	✓	✓	✓		✓	
East US 2 ¹	✓	✓	✓	✓	✓	✓
Mexico Central		✓				
North Central US	✓		✓		✓	✓

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
1						
South Central US	✓	✓	✓		✓	✓
1						
West US 1	✓		✓		✓	✓
3	✓	✓	✓		✓	✓
West US 3	✓	✓	✓		✓	✓
West Central US	✓		✓		✓	
1						

¹ This region supports [agentic retrieval](#) and [semantic ranker](#) on the free tier.

² This region is experiencing capacity constraints that prevent the creation of new search services and might present failures for scale operations. Please choose a different region.

³ This region doesn't have indexer support for [Microsoft Purview sensitivity labels](#).

Europe

 Expand table

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
France Central	✓	✓	✓		✓	✓
1						
Germany West Central ¹	✓	✓	✓		✓	
Italy North		✓	✓	✓	✓	
Norway East	✓	✓		✓		
North Europe	✓	✓	✓		✓	✓
Poland Central			✓		✓	
1						
Spain Central ²		✓	✓		✓	✓

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Sweden Central ¹	✓	✓	✓		✓	✓
Switzerland North ¹	✓	✓	✓	✓	✓	✓
Switzerland West	✓	✓	✓		✓	
UK South ¹	✓	✓	✓	✓	✓	✓
UK West			✓		✓	
West Europe ¹	✓	✓	✓	✓	✓	✓

¹ This region supports [agentic retrieval](#) and [semantic ranker](#) on the free tier.

² [Higher storage limits](#) aren't available in this region. If you want higher limits, choose a different region.

Middle East

Expand table

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Israel Central ¹		✓				
Qatar Central ¹		✓	✓		✓	
UAE North ^{2, 3}	✓	✓	✓	✓	✓	

¹ [Higher storage limits](#) aren't available in this region. If you want higher limits, choose a different region.

² This region supports [agentic retrieval](#) and [semantic ranker](#) on the free tier.

³ This region is experiencing capacity constraints that prevent the creation of new search services. Please choose a different region.

Africa

 Expand table

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
South Africa North ¹						

¹ This region supports [agentic retrieval](#) and [semantic ranker](#) on the free tier.

Asia Pacific

 Expand table

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Australia East ¹						
Australia Southeast						
Central India						
East Asia						
Indonesia Central						
Jio India West						
Jio India Central						
Japan East ¹						
Japan West						
Korea Central ¹						
Korea South						

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Malaysia West		✓				
New Zealand North		✓				
South India		✓				
Southeast Asia	✓	✓	✓		✓	✓

¹ This region supports [agentic retrieval](#) and [semantic ranker](#) on the free tier.

Azure Government regions

[Expand table](#)

Region	AI enrichment	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
Arizona	✓		✓		✓	✓
Texas						
Virginia	✓	✓	✓		✓	✓

Azure operated by 21Vianet

[Expand table](#)

Region	AI enrichment ¹	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
China East						
China East 2 ²	✓					
China East 3						
China						

Region	AI enrichment ¹	Availability zones	Agentic retrieval	Confidential computing	Semantic ranker	Query rewrite
North						
China						
North 2 ²						
China		✓	✓		✓	✓
North 3						

¹ Only China East 2 fully supports AI enrichment. In other 21Vianet regions, you can use skillsets with the [Azure OpenAI Embedding skill](#) for integrated vectorization, which depends on the availability of Azure OpenAI and Azure AI Search in your region. Otherwise, AI enrichment isn't supported.

² [Higher storage limits](#) aren't available in this region. If you want higher limits, choose a different region.

Related content

- [Azure Vision region list](#)
- [Availability zone region availability](#)
- [Azure product by region page ↗](#)

Last updated on 11/19/2025

Choose a service tier for Azure AI Search

Part of [creating a search service](#) is choosing a pricing tier (or SKU). In the Azure portal, tier is specified in the [Select Pricing Tier](#) page when you create the service. In PowerShell or Azure CLI, the tier is specified through the `-Sku` parameter.

The tier determines the:

- Maximum number of indexes and other objects allowed on the service.
- Size and speed of partitions (physical storage).
- Billable rate as a fixed monthly cost, but also an incremental cost if you add capacity.
- Workload characteristics. Some tiers are optimized for specific workloads.

In a few instances, the tier you choose determines the availability of [premium features](#).

Billing rates are shown in the Azure portal's [Select Pricing Tier](#) page. You can check the [pricing page](#) for regional rates and review [Plan and manage costs](#) to learn more about the billing model.

(!) Note

Search services created after April 3, 2024 have larger partitions and higher vector quotas at almost every tier. For more information, see [Service limits](#).

Tier descriptions

Tiers include **Free**, **Basic**, **Standard**, and **Storage Optimized**. Standard and Storage Optimized are available with several configurations and capacities. The following screenshot from Azure portal shows the available tiers, minus pricing (which you can find in the Azure portal and on the [pricing page](#)).

Select Pricing Tier									X
Browse available skus and their features									
Sku	Offering	Indexes	Indexers	Vector quota	Total storage	Search units	Replicas	Partitions	Search unit cost/month (est.)
F	Free	3	3	25 MB ⓘ	50 MB	1	1	1	
B	Basic	15	15	5 GB	15 GB/Partition	9	3	3	
S	Standard	50	50	35 GB/Partition	160 GB/Partition	36	12	12	
S2	Standard	200	200	100 GB/Partition	350 GB/Partition	36	12	12	
S3	Standard	200	200	200 GB/Partition	700 GB/Partition	36	12	12	
S3HD	High-density	1000	0	200 GB/Partition	700 GB/Partition	36	12	3	
L1	Storage Optimized	10	10	12 GB/Partition	1 TB/Partition	36	12	12	
L2	Storage Optimized	10	10	36 GB/Partition	2 TB/Partition	36	12	12	

Estimated costs not included in this screenshot

Higher storage limits are available for new services in this region at no additional cost.

Free creates a [limited search service](#) for smaller projects, like running tutorials and code samples. Internally, system resources are shared among multiple subscribers. You can't scale a free service, run significant workloads, and some premium features aren't available. You can only have one free search service per Azure subscription. If the service is inactive for an extended period of time, it might be deleted to free up capacity, especially if the region is under capacity constraints.

The most commonly used billable tiers include:

- **Basic** has the ability to meet SLA with its support for three replicas.
- **Standard (S1, S2, S3)** is the default. It gives you more flexibility in scaling for workloads. You can scale both partitions and replicas. With dedicated resources under your control, you can deploy larger projects, optimize performance, and increase capacity.

Some tiers are designed for certain types of work:

- **Standard 3 High Density (S3 HD)** is a *hosting mode* for S3, where the underlying hardware is optimized for a large number of smaller indexes and is intended for multitenancy scenarios. S3 HD has the same per-unit charge as S3, but the hardware is optimized for fast file reads on a large number of smaller indexes.
- **Storage Optimized (L1, L2)** tiers offer larger storage capacity at a lower price per TB than the Standard tiers. These tiers are designed for large indexes that don't change very often. The primary tradeoff is higher query latency, which you should validate for your specific application requirements.

You can find out more about the various tiers on the [pricing page](#), in the [Service limits in Azure AI Search](#) article, and on the Azure portal page when you're provisioning a service.

Region availability by tier

The [regions list](#) provides the locations where Azure AI Search is offered. Some regions might have capacity constraints for certain tiers, which prevents the creation of new search services on those tiers. The list uses footnotes to indicate constrained regions and tiers.

When you create a search service in the Azure portal, unavailable region–tier combinations are automatically excluded.

Feature availability by tier

Most features are available on all tiers, including the Free tier. In a few cases, the tier determines the availability of a feature. The following table describes the constraints.

Feature	Tier considerations
Indexers	Indexers aren't available on S3 HD. Indexers have more limitations on the free tier.
indexer executionEnvironment configuration parameter	The ability to pin all indexer processing to just the search clusters allocated to your search service requires S2 and higher.
AI enrichment	Runs on the Free tier but not recommended for large workloads.
Managed or trusted identities for outbound (indexer) access	Not available on the Free tier.
Customer-managed encryption keys	Not available on the Free tier.
IP firewall access	Not available on the Free tier.
Private endpoint (integration with Azure Private Link)	For inbound connections to a search service, not available on the Free tier. For outbound connections by indexers to other Azure resources, not available on Free or S3 HD. For indexers that use skillsets, not available on Free, Basic, S1, or S3 HD.
Availability zones	Not available on the Free tier.
Semantic ranker	Runs on the Free tier but not recommended for large workloads.

Resource-intensive features might not work well unless you give it sufficient capacity. For example, [AI enrichment](#) has long-running skills that time out on a Free service unless the dataset is small.

Upper limits

Tiers determine the maximum storage of the service itself, plus the maximum number of indexes, indexers, data sources, skillsets, and synonym maps that you can create. For a full break out of all limits, see [Service limits in Azure AI Search](#).

Partition size and speed

Tier pricing includes details about per-partition storage that ranges from 15 GB for Basic, up to 2 TB for Storage Optimized (L2) tiers. Other hardware characteristics, such as speed of operations, latency, and transfer rates, aren't published, but tiers that are designed for specific

solution architectures are built on hardware that has the features to support those scenarios. For more information about partitions, see [Estimate and manage capacity](#) and [Reliability in Azure AI Search](#).

(!) Note

Higher-capacity partitions became available in select regions in April 2024. A second wave of higher-capacity partitions was released in May 2024. If you have an older search service, you might be able to [upgrade your service](#) to benefit from more capacity at the same billing rate.

Billing rates

Tiers have different billing rates, with higher rates for tiers that run on more expensive hardware or provide more expensive features. The tier billing rate can be found in the [Azure pricing pages](#) for Azure AI Search.

Once you create a service, the billing rate becomes both a *fixed cost* of running the service around the clock, and an *incremental cost* if you choose to add more capacity.

Search services are allocated computing resources in the form of *partitions* (for storage), and *replicas* (instances of the query engine). Initially, a service is created with one of each, and the billing rate is inclusive of both resources. However, if you scale capacity, the costs go up or down in increments of the billable rate.

The following example provides an illustration. Assume a hypothetical billing rate of \$100 per month. If you keep the search service at its initial capacity of one partition and one replica, then \$100 is what you can expect to pay at the end of the month. However, if you add two more replicas to achieve high availability, the monthly bill increases to \$300 (\$100 for the first replica-partition pair, followed by \$200 for the two replicas).

This billing model is based on the concept of applying the billing rate to the number *search units* (SU) used by a search service. All services are initially provisioned at one SU, but you can increase the SUs by adding either partitions or replicas to handle larger workloads. For more information, see [How to estimate costs of a search service](#).

Tier changes

(!) Note

Existing search services can switch between Basic and Standard (S1, S2, and S3) tiers. Your current service configuration can't exceed the limits of the target tier, and your region can't have capacity constraints on the target tier. For more information, see [Change your pricing tier](#).

To switch to a different tier than those previously listed:

1. [Create a search service](#) on the new tier.
2. Deploy your search content onto the new service. [Follow this checklist](#) to ensure you have all the content.
3. Delete the old service when you're sure it's no longer needed.

For large indexes that you don't want to rebuild from scratch, use one of the following backup and restore samples:

- [Backup and restore sample \(C#\)](#) ↗
- [Backup and restore sample \(Python\)](#) ↗
- [Backup and restore sample for very large indexes \(Python\)](#) ↗

Next steps

The best way to choose a pricing tier is to start with a least-cost tier, and then allow experience and testing to inform your decision to keep the service or switch to a higher tier.

For next steps, we recommend that you create a search service at a tier that can accommodate the level of testing you propose to do, and then review the following guidance on estimating cost and capacity:

- [Create a search service](#)
- [Estimate costs](#)
- [Estimate capacity](#)

Upgrade your Azure AI Search service in the Azure portal

08/01/2025

An upgrade brings older search services to the capabilities of new services created in the same region. Specifically, it upgrades the computing power of the underlying service. This one-time operation doesn't introduce breaking changes to your application, and you shouldn't need to change any code.

For [eligible services](#), an upgrade increases the [partition storage](#) and [vector index size](#) on the same tier at no extra cost.

💡 Tip

Looking to [change your pricing tier](#)? You can switch between Basic and Standard (S1, S2, and S3) tiers.

This article describes how to upgrade your service in the [Azure portal](#). Alternatively, you can use the [Search Management REST APIs](#) to upgrade your service programmatically. For more information, see [Manage your search service using REST](#).

About service upgrades

In April 2024, Azure AI Search increased the [storage capacity](#) of newly created search services. Services created before April 2024 saw no capacity changes, so if you wanted larger and faster partitions, you had to create a new service. However, some older services can now be upgraded to benefit from the higher-capacity partitions.

Currently, an upgrade only increases the [storage limit](#) and [vector index size](#) of eligible services.

Upgrade eligibility

To qualify for an upgrade, your service must:

- ✓ Have been [created before April 3, 2024](#). Services created after this date should already have higher capacity.
- ✓ Be in a [region where higher capacity is enabled](#). Most regions provide higher-capacity partitions, as noted in the table's footnotes.
- ✓ Be in a [region that doesn't have capacity constraints on your pricing tier](#). Constrained regions and tiers are noted in the footnotes of each table.

Important

Some search services created before January 1, 2019 don't support upgrades. In this situation, you must create a new service in a [high-capacity region](#) to get increased storage and vector limits.

Higher storage limits

For [eligible services](#), the following table compares the storage limit (per partition) before and after an upgrade.

 [Expand table](#)

	Basic ¹	S1	S2	S3/HD	L1	L2
Limit before upgrade	2 GB	25 GB	100 GB	200 GB	1 TB	2 TB
Limit after upgrade	15 GB	160 GB	512 GB	1 TB	2 TB	4 TB

¹ Basic services created before April 3, 2024 were originally limited to one partition, which increases to three partitions after an upgrade. [Partition counts for all other pricing tiers](#) stay the same.

Higher vector limits

For [eligible services](#), the following table compares the vector index size (per partition) before and after an upgrade.

 [Expand table](#)

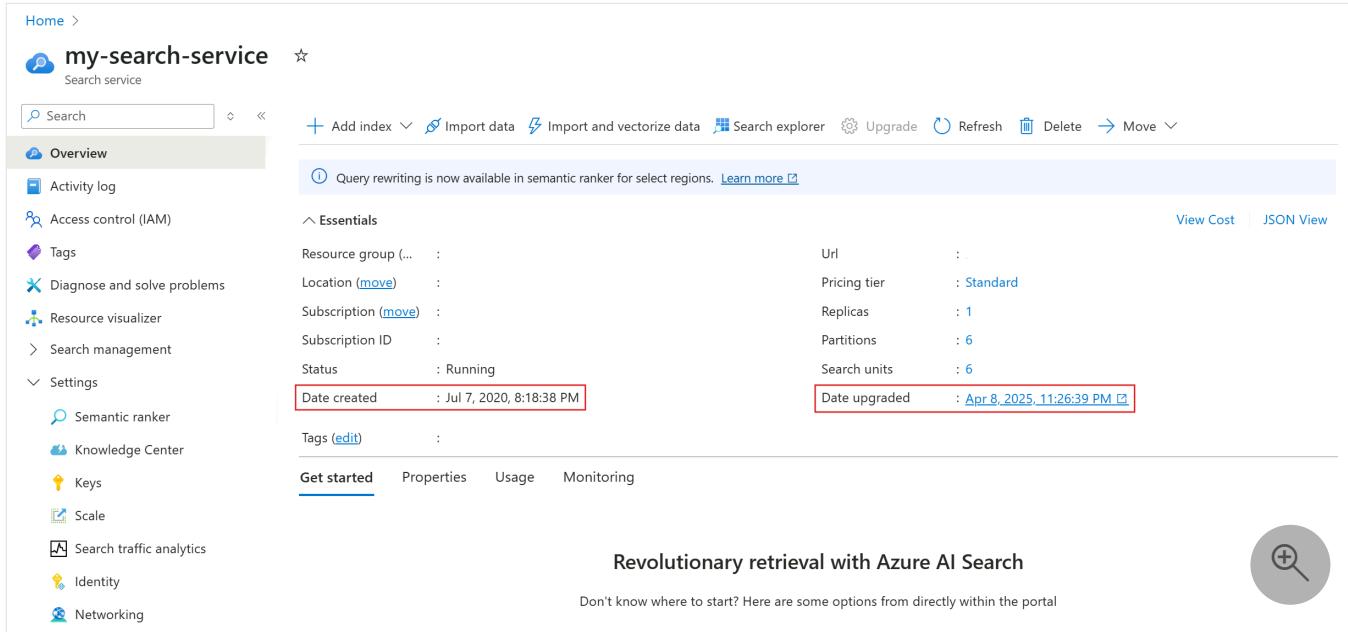
	Basic	S1	S2	S3/HD	L1	L2
Limit before upgrade	0.5 GB ¹ or 1 GB ²	1 GB ¹ or 3 GB ²	6 GB ¹ or 12 GB ²	12 GB ¹ or 36 GB ²	12 GB	36 GB
Limit after upgrade	5 GB	35 GB	150 GB	300 GB	150 GB	300 GB

¹ Applies to services created before July 1, 2023.

² Applies to services created between July 1, 2023 and April 3, 2024 in all regions except Germany West Central, Qatar Central, and West India, to which the ¹ limits apply.

Check your service creation or upgrade date

On the **Overview** page, you can view various metadata about your search service, including **Date created** and **Date upgraded**.



The screenshot shows the Azure AI Search service overview page for 'my-search-service'. The 'Overview' tab is selected. In the 'Essentials' section, two fields are highlighted with red boxes: 'Date created' (Jul 7, 2020, 8:18:38 PM) and 'Date upgraded' (Apr 8, 2025, 11:26:39 PM). The page also includes a sidebar with various service management options like Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Search management, Settings, Semantic ranker, Knowledge Center, Keys, Scale, Search traffic analytics, Identity, and Networking.

The date you created your service partially determines its [upgrade eligibility](#). If your service has never been upgraded, **Date upgraded** doesn't appear.

Upgrade your service

You can't undo a service upgrade. Before you proceed, make sure that you want to permanently increase the [storage limit](#) and [vector index size](#) of your search service. We recommend that you test this operation in a nonproduction environment.

The availability of your search service during an upgrade depends on how many replicas you've provisioned. With two or more replicas, your service remains available while one replica is updated. For more information, see [Reliability in Azure AI Search](#).

To upgrade your service:

1. Sign in to the [Azure portal](#) and select your search service.
2. On the **Overview** page, select **Upgrade** from the command bar.

The screenshot shows the Azure portal interface for a search service named 'my-search-service'. The top navigation bar includes options like 'Add index', 'Import data', 'Import and vectorize data', 'Search explorer', 'Upgrade' (which is highlighted with a red box), 'Refresh', 'Delete', and 'Move'. Below the navigation bar, there are two status messages: one about query rewriting being available in semantic ranker for select regions, and another about an upgrade being available for the service. The main content area is titled 'Overview' and contains sections for 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Resource visualizer', and 'Search management'. A 'View Cost' and 'JSON View' link is also present. On the right side, there's a search bar with a magnifying glass icon.

If this button appears dimmed, an upgrade isn't available for your service. Your service either [has the latest upgrade](#) or [doesn't qualify for an upgrade](#).

3. Review the upgrade details for your service, and then select **Upgrade**.

The screenshot shows a modal dialog box titled 'Upgrade a search service'. It contains the following text:
The following search service will be permanently upgraded, and this operation can't be undone.
Search service name: (with a cloud icon)
Created or last upgraded on: Aug 17, 2023, 1:07:27 PM
After the search service is upgraded, you'll have a **higher total storage limit**.
[Learn more](#) about the upgrade available for your service based on when it was created or last upgraded.
At the bottom, there are two buttons: 'Upgrade' (highlighted with a red box) and 'Cancel'. There is also a search bar with a magnifying glass icon in the bottom right corner.

A confirmation appears reminding you that the upgrade can't be undone.

4. To permanently upgrade your service, select **Upgrade**.

Upgrade confirmation

Upgrading the search service is a permanent action and can't be undone.

Upgrade

Go back



5. Check your notifications to confirm that the operation started.

Depending on the size of your service, this operation can take several hours to complete. If the upgrade fails, your service returns to its original state.

Next step

After you upgrade your search service, you might want to reconsider your scale configuration:

[Estimate and manage capacity of an Azure AI Search service](#)

Service limits in Azure AI Search

Maximum limits on storage, workloads, and quantities of indexes and other objects depend on the [pricing tier](#) of your Azure AI Search service:

- **Free** is a multitenant shared service that comes with your Azure subscription.
- **Basic** provides dedicated computing resources for production workloads at a smaller scale.
- **Standard** runs on dedicated machines with more storage and processing capacity at every level. Standard comes in four levels: S1, S2, S3, and S3 HD. S3 High Density (S3 HD) is engineered for [multi-tenancy](#) and large quantities of small indexes (3,000 indexes per service). S3 HD doesn't support [indexers](#), so data ingestion must use APIs that push data from the source to the index.
- **Storage Optimized** runs on dedicated machines with more total storage, storage bandwidth, and memory than **Standard**. This tier targets large, slow-changing indexes. Storage Optimized comes in two levels: L1 and L2.

Subscription limits

You can create multiple *billable* search services (Basic and higher), up to the maximum number of services allowed at each tier, per region. For example, you could create up to 16 services at the Basic tier and another 16 services at the S1 tier within the same subscription and region. You could then create an additional 16 Basic services in another region for a combined total of 32 Basic services under the same subscription. For more information about tiers, see [Choose a tier \(or SKU\) for Azure AI Search](#).

Maximum service limits can be raised upon request. If you need more services within the same subscription, [file a support request](#).

[+] [Expand table](#)

Resource	Free ¹	Basic	S1	S2	S3	S3 HD	L1	L2
Maximum services per region	1	16	16	8	6	6	6	6
Maximum search units (SU) ²	N/A	3 SU	36 SU	36 SU	36 SU	36 SU	36 SU	36 SU

¹ You can have one free search service per Azure subscription. The free tier is based on infrastructure shared with other customers. Because the hardware isn't dedicated, scale-up isn't supported, and storage is limited to 50 MB. A free search service might be deleted after extended periods of inactivity to make room for more services.

² Search units (SU) are billing units, allocated as either a *replica* or a *partition*. You need both. To learn more about SU combinations, see [Estimate and manage capacity of a search service](#).

Service limits

The following table covers SLA, partition counts, and replica counts at the service level.

[+] [Expand table](#)

Resource	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Service level agreement (SLA)	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Partitions	N/A	3 ¹	12	12	12	3	12	12
Replicas	N/A	3	12	12	12	12	12	12

¹ Basic tier supports three partitions and three replicas, for a total of nine search units (SU) on [new search services](#) created after April 3, 2024. Older basic services are limited to one partition and three replicas.

A search service is subject to a maximum storage limit (partition size multiplied by the number of partitions) or by a hard limit on the [maximum number of indexes](#) or [indexers](#), whichever comes first.

Service-level agreements (SLAs) apply to billable services that have two or more replicas for query workloads, or three or more replicas for query and indexing workloads. The number of partitions isn't an SLA consideration. For more information, see [Reliability in Azure AI Search](#).

Free services don't have fixed partitions or replicas and share resources with other subscribers.

Partition storage (GB)

Per-service storage limits vary by two things: [service creation date](#) and [region](#). There are higher limits for [newer services](#) in most supported regions.

This table shows the progression of storage quota increases in GB over time. Starting in April 2024, higher capacity partitions were brought online in the regions listed in the footnotes. If you have an older service in a supported region, check if you can [upgrade your service](#) to the higher storage limits.

[+] [Expand table](#)

Service creation date	Basic	S1	S2	S3/HD	L1	L2
Before April 3, 2024	2	25	100	200	1,024	2,048
April 3, 2024 through May 17, 2024 ¹	15	160	512	1,024	1,024	2,048
After May 17, 2024 ²	15	160	512	1,024	2,048	4,096
After February 10, 2025 ³	15	160	512	1,024	2,048	4,096

¹ Higher capacity storage for Basic, S1, S2, S3 in these regions. **Americas:** Brazil South, Canada Central, Canada East, East US, East US 2, Central US, North Central US, South Central US, West US, West US 2, West US 3, West Central US. **Europe:** France Central, Italy North, North Europe, Norway East, Poland Central, Switzerland North, Sweden Central, UK South, UK West. **Middle East:** UAE North. **Africa:** South Africa North. **Asia Pacific:** Australia East, Australia Southeast, Central India, Jio India West, East Asia, Southeast Asia, Japan East, Japan West, Korea Central, Korea South.

² Higher capacity storage for L1 and L2. More regions provide higher capacity at every billable tier. **Americas:** East US 2 EUAP. **Europe:** Germany North, Germany West Central, Switzerland West. **Azure Government:** Texas, Arizona, Virginia. **Africa:** South Africa North. **Asia Pacific:** China North 3, China East 3.

³ Higher capacity storage is available in West Europe.

ⓘ Important

Currently, higher storage limits aren't available in the following regions, which are subject to the pre-April 3 limits.

- Israel Central
- Qatar Central
- Spain Central
- South India

Index limits

Expand table

Resource	Free	Basic ¹	S1	S2	S3	S3 HD	L1	L2
Maximum indexes	3	5 or 15	50	200	200	1000 per partition or 3000 per service	10	10

Resource	Free	Basic ¹	S1	S2	S3	S3 HD	L1	L2
Maximum simple fields per index ²	1000	100	1000	1000	1000	1000	1000	1000
Maximum dimensions per vector field	4096	4096	4096	4096	4096	4096	4096	4096
Maximum complex collections per index	40	40	40	40	40	40	40	40
Maximum elements across all complex collections per document ³	3000	3000	3000	3000	3000	3000	3000	3000
Maximum depth of complex fields	10	10	10	10	10	10	10	10
Maximum suggesters per index	1	1	1	1	1	1	1	1
Maximum scoring profiles per index	100	100	100	100	100	100	100	100
Maximum semantic configurations per index	100	100	100	100	100	100	100	100
Maximum functions per profile	8	8	8	8	8	8	8	8
Maximum index size ⁴	N/A	N/A	N/A	1.88 TB	2.34 TB	100 GB	N/A	N/A

¹ Basic services created before December 2017 have lower limits (5 instead of 15) on indexes. Basic tier is the only tier with a lower limit of 100 fields per index.

² The upper limit on fields includes both first-level fields and nested subfields in a complex collection. For example, if an index contains 15 fields and has two complex collections with five subfields each, the field count of your index is 25. Indexes with a very large fields collection can be slow. [Limit fields and attributes](#) to just those you need, and run indexing and query test to ensure performance is acceptable.

³ An upper limit exists for elements because having a large number of them significantly increases the storage required for your index. An element of a complex collection is defined as a member of that collection. For example, assume a [Hotel document with a Rooms complex collection](#). Each room in the Rooms collection is considered an element. During indexing, the indexing engine can safely process a maximum of 3,000 elements across the document as a whole. [This limit](#) was introduced in `api-version=2019-05-06` and applies to complex collections only, and not to string collections or to complex fields.

⁴ For most tiers, the maximum index size is the total available storage on your search service. For S2, S3, and S3 HD services with multiple partitions, and therefore more storage, the maximum size of a single index is provided in the table. Applies to search services created after April 3, 2024.

You might find some variation in maximum limits if your service happens to be provisioned on a more powerful cluster. The limits here represent the common denominator. Indexes built to the above specifications are portable across equivalent service tiers in any region.

Document limits

Maximum number of documents per index are:

- 24 billion on Basic, S1, S2, S3
- 2 billion on S3 HD
- 288 billion on L1
- 576 billion on L2

Maximum size of each document is approximately 16 megabytes. Document size is actually a limit on the size of the indexing API request payload, which is 16 megabytes. That payload can be a single document, or a batch of documents. For a batch with a single document, the maximum document size is 16 MB of JSON.

Document size applies to *push mode* indexing that uploads documents to a search service. If you're using an indexer for *pull mode* indexing, your source files can be any file size, subject to [indexer limits](#). For the blob indexer, file size limits are larger for higher tiers. For example, the S1 limit is 128 megabytes, S2 limit is 256 megabytes, and so forth.

When you estimate document size, remember to index only the fields that add value to your search scenarios. Exclude source fields that have no purpose in the queries you intend to run.

Vector index size limits

When you index documents with vector fields, Azure AI Search constructs internal vector indexes using the algorithm parameters you provide. The size of these vector indexes is restricted by the memory reserved for vector search for your service's tier (or [SKU](#)). For guidance on managing and maximizing vector storage, see [Vector index size and staying under limits](#).

Vector limits vary by:

- [Service creation date](#)
- [Region](#)

Higher vector limits from April 2024 onwards exist on *new search services* in regions providing the extra capacity, which is most of them. If you have an older service in a supported region, check if

you can [upgrade your service](#) to the higher vector limits.

This table shows the progression of vector quota increases in GB over time. The quota is per partition, so if you scale a new Standard (S1) service to 6 partitions, the total vector quota is 35 multiplied by 6.

[+] [Expand table](#)

Service creation date	Basic	S1	S2	S3/HD	L1	L2
Before July 1, 2023 ¹	0.5	1	6	12	12	36
July 1, 2023 through April 3, 2024 ²	1	3	12	36	12	36
April 3, 2024 through May 17, 2024 ³	5	35	150	300	12	36
After May 17, 2024 ⁴	5	35	150	300	150	300

¹ Initial vector limits during early preview.

² Vector limits during the later preview period. Three regions didn't have the higher limits: Germany West Central, West India, Qatar Central.

³ Higher vector quota based on the larger partitions for supported tiers and regions.

⁴ Higher vector quota for more tiers and regions based on partition size updates.

The service enforces a vector index size quota *for every partition* in your search service. Each extra partition increases the available vector index size quota. This quota is a hard limit to ensure your service remains healthy, which means that further indexing attempts once the limit is exceeded results in failure. You can resume indexing once you free up available quota by either deleting some vector documents or by scaling up in partitions.

ⓘ Important

Higher vector limits are tied to [larger partition sizes](#). Currently, higher vector limits aren't available in the following regions, which are subject to the July–April limits.

- Israel Central
- Qatar Central
- Spain Central
- South India

Indexer limits

Maximum running times exist to provide balance and stability to the service as a whole, but larger data sets might need more indexing time than the maximum allows. If an indexing job can't complete within the maximum time allowed, try running it on a schedule. The scheduler keeps track of indexing status. If a scheduled indexing job is interrupted for any reason, the indexer can pick up where it last left off at the next scheduled run.

[\[+\] Expand table](#)

Resource	Free ¹	Basic ²	S1	S2	S3	S3 HD ³	L1	L2
Maximum indexers	3	5 or 15	50	200	200	N/A	10	10
Maximum datasources	3	5 or 15	50	200	200	N/A	10	10
Maximum skillsets ⁴	3	5 or 15	50	200	200	N/A	10	10
Maximum indexing load per invocation	10,000 documents	Limited only by maximum documents	Limited only by maximum documents	Limited only by maximum documents	Limited only by maximum documents	N/A	No limit	No limit
Minimum schedule	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes
Maximum running time ⁵	1-3 or 3-10 minutes	2 or 24 hours	N/A	2 or 24 hours	2 or 24 hours			
Blob indexer: maximum blob size, MB	16	16	128	256	256	N/A	256	256
Blob indexer: maximum characters of content extracted from a blob	256,000	512,000	4 million	8 million	16 million	N/A	4 million	4 million

¹ Free services have indexer maximum execution time of 3 minutes for blob sources and 1 minute for all other data sources. Indexer invocation is once every 180 seconds. For AI indexing that calls Foundry Tools, free services are limited to 20 free transactions per indexer per day, where a

⁶

transaction is defined as a document that successfully passes through the enrichment pipeline.

(Tip: You can reset an indexer to reset its count.)

² Basic services created before December 2017 have lower limits (5 instead of 15) on indexers, data sources, and skillsets.

³ S3 HD services don't include indexer support.

⁴ Maximum of 30 skills per skillset.

⁵ Regarding the 2 or 24 hour maximum duration for indexers: a 2-hour maximum is the most common and it's what you should plan for. It refers to indexers that run in the [public environment](#), which offloads computationally intensive processing and leaves more resources for queries. The 24-hour limit applies if you configure the indexer to run in a private environment using only the infrastructure that's allocated to your search service. Some older indexers are incapable of running in the public environment, and those indexers always have a 24-hour processing range. If you have unscheduled indexers that run continuously for 24 hours, you can assume those indexers couldn't be migrated to the newer infrastructure. As a general rule, for indexing jobs that can't finish within two hours, put the indexer on a [5-minute schedule](#) so that the indexer can quickly pick up where it left off. On the Free tier, the 3-10 minute maximum running time is for indexers with skillsets.

⁶ The maximum number of characters is based on Unicode code units, specifically UTF-16.

Note

As stated in [Index limits](#), indexers also enforce the upper limit of 3,000 elements across all complex collections per document starting with the latest GA API version that supports complex types ([2019-05-06](#)) onwards. This means that if you created your indexer with a prior API version, you won't be subject to this limit. To preserve maximum compatibility, an indexer that was created with a prior API version and then updated with an API version [2019-05-06](#) or later, will still be **excluded** from the limits. Customers should be aware of the adverse impact of having very large complex collections (as stated previously) and we highly recommend creating any new indexers with the latest GA API version.

Shared private link resource limits

Indexers can access other Azure resources [over private endpoints](#) managed via the [shared private link resource API](#). This section describes the limits associated with this capability.

 Expand table

Resource	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Private endpoint indexer support	No	Yes	Yes	Yes	Yes	No	Yes	Yes
Private endpoint support for indexers with a skillset ¹	No	No	Yes	Yes	Yes	No	Yes	Yes
Private endpoint support for skillsets with an embedding skill ²	No	Yes	Yes	Yes	Yes	No	Yes	Yes
Maximum private endpoints	N/A	10 or 30	100	400	400	N/A	20	20
Maximum distinct resource types ³	N/A	4	7	15	15	N/A	4	4

¹ AI enrichment and image analysis are computationally intensive and consume disproportionate amounts of available processing power. For this reason, private connections are disabled on lower tiers to ensure the performance and stability of the search service itself. On Basic services, private connections to a Microsoft Foundry resource are unsupported to preserve service stability. For the S1 tier, make sure the service was created with [higher limits](#) after April 3, 2024. Indexers with more than 2 Azure OpenAI Embedding or Azure Vision multimodal embeddings skills are restricted from running in private environment, and private connections aren't available.

² Private connections to an embedding model are supported on Basic and S1 high-capacity search services created after April 3, 2024, with the [higher limits](#) for storage and computational processing.

³ The number of distinct resource types are computed as the number of unique `groupId` values used across all shared private link resources for a given search service, irrespective of the status of the resource.

Synonym limits

Maximum number of synonym maps varies by tier. Each rule can have up to 20 expansions, where an expansion is an equivalent term. For example, given "cat", association with "kitty", "feline", and "felis" (the genus for cats) would count as 3 expansions.

[+] [Expand table](#)

Resource	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Maximum synonym maps	3	3	5	10	20	20	10	10
Maximum number of rules per map	5000	20000	20000	20000	20000	20000	20000	20000

Index alias limits

Maximum number of [index aliases](#) varies by tier and [service creation date](#). On all tiers, if the service was created after October 2022, the maximum number of aliases is double the maximum number of indexes allowed. If the service was created before October 2022, the limit is the number of indexes allowed.

[Expand table](#)

Service creation date	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Before October 2022	3	5 or 15 ¹	50	200	200	1000 per partition or 3000 per service	10	10
After October 2022	6	30	100	400	400	2000 per partition or 6000 per service	20	20

¹ Basic services created before December 2017 have lower limits (5 instead of 15) on indexes.

Agentic retrieval limits

Each [knowledge base](#) contains [knowledge sources](#), which are data source connections, and configurations that agents consume for [agentic retrieval](#). The following limits apply to knowledge sources and knowledge bases per service tier.

[Expand table](#)

Resource	Free	Basic ¹	S1	S2	S3	S3 HD	L1	L2
Maximum knowledge sources	3	5 or 15	50	200	200	0	10	10
Maximum knowledge bases	3	5 or 15	50	200	200	0	10	10

¹ Basic services created before April 3, 2024 have lower limits (5 instead of 15) on knowledge sources and knowledge bases.

Data limits (AI enrichment)

Data limits apply to an [AI enrichment pipeline](#) that makes calls to Azure Language in Foundry Tools for [entity recognition](#), [entity linking](#), [key phrase extraction](#), [sentiment analysis](#), [language detection](#), and [personal-information detection](#). The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the sentiment analyzer, use the [Text Split skill](#).

Throttling limits

API requests are throttled as the system approaches peak capacity. Throttling behaves differently for different APIs. Query APIs (Search/Suggest/Autocomplete) and indexing APIs throttle

dynamically based on the load on the service. Index APIs and service operations API have static request rate limits.

Static rate request limits for operations related to an index:

- List Indexes (GET /indexes): 3 per second per search unit
- Get Index (GET /indexes/myindex): 10 per second per search unit
- Create Index (POST /indexes): 12 per minute per search unit
- Create or Update Index (PUT /indexes/myindex): 6 per second per search unit
- Delete Index (DELETE /indexes/myindex): 12 per minute per search unit

Static rate request limits for operations related to a service:

- Service Statistics (GET /servicestats): 4 per second per search unit

Semantic ranker throttling limits

[Semantic ranker](#) uses a queuing system to manage concurrent requests. This system allows search services get the highest number of queries per second possible. When the limit of concurrent requests is reached, additional requests are placed in a queue. If the queue is full, further requests are rejected and must be retried.

Total semantic ranker queries per second varies based on the following factors:

- The tier of the search service. Both queue capacity and concurrent request limits vary by tier.
- The number of search units in the search service. The simplest way to increase the maximum number of concurrent semantic ranker queries is to [add more search units to your search service](#).
- The total available semantic ranker capacity in the region.
- The amount of time it takes to serve a query using semantic ranker. This varies based on how busy the search service is.

The following table describes the semantic ranker throttling limits by tier, subject to available capacity in the region. You can contact Microsoft support to request a limit increase.

 [Expand table](#)

Resource	Basic	S1	S2	S3	S3 HD	L1	L2
Maximum concurrent requests (per search unit)	2	3	4	4	4	4	4
Maximum request queue size (per search unit)	4	6	8	8	8	8	8

API request limits

Limits on queries exist because unbounded queries can destabilize your search service. Typically, such queries are created programmatically. If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

Limits on payloads exist for similar reasons, ensuring the stability of your search service. The limit applies to the entire request, inclusive of all its components. For example, if the request batches several documents or commands, the entire request must fit within the supported limit.

If you must exceed a supported limit, you should [test your workload](#) so that you know what to expect.

Except where noted, the following API requests apply to all programmable interfaces, including the Azure SDKs.

General:

- Supported maximum payload limit is 16 MB for indexing and query request via REST API and SDKs.
- Maximum 8-KB URL length (applies to REST APIs only).

Indexing APIs:

- Supported maximum 1,000 documents per batch of index uploads, merges, or deletes.

Query APIs:

- Maximum 10 fields in a vector query
- Maximum 32 fields in \$orderby clause.
- Maximum 100,000 characters in a search clause.
- Maximum number of clauses in search is 3,000.
- Maximum limits on [wildcard](#) and [regular expression](#) queries, as enforced by [Lucene](#). It caps the number of patterns, variations, or matches to 1,000 instances. This limit is in place to avoid engine overload.

Search terms:

- Supported maximum search term size is 32,766 bytes (32 KB minus 2 bytes) of UTF-8 encoded text. Applies to keyword search and the text property of vector search.
- Supported maximum search term size is 1,000 characters for [prefix search](#) and [regex search](#).

API response limits

- Maximum 1,000 documents returned per page of search results
- Maximum 100 suggestions returned per Suggest API request

The search engine returns 50 results by default, but you can [override this parameter](#) up to the maximum limit.

API key limits

API keys are used for service authentication. There are two types. Admin keys are specified in the request header and grant full read-write access to the service. Query keys are read-only, specified on the URL, and typically distributed to client applications.

- Maximum of 2 admin keys per service
- Maximum of 50 query keys per service

Last updated on 11/20/2025

Estimate and manage capacity of a search service

In Azure AI Search, capacity is based on *replicas* and *partitions* that can be scaled to your workload. Replicas are copies of the search engine. Partitions are units of storage. Each new search service starts with one each, but you can add or remove replicas and partitions independently to accommodate fluctuating workloads. Adding capacity increases the [cost of running a search service](#).

The physical characteristics of replicas and partitions, such as processing speed and disk IO, vary by [pricing tier](#). On a standard search service, the replicas and partitions are faster and larger than those of a basic service.

Changing capacity isn't instantaneous. It can take up to an hour to commission or decommission partitions, especially on services with large amounts of data.

When scaling a search service, you can choose from the following tools and approaches:

- [Azure portal](#)
- [Azure PowerShell](#)
- [Azure CLI](#)
- [Management REST API](#)

(!) Note

If your service was created before April or May 2024, a one-time upgrade to higher storage limits might be available at no extra cost. For more information, see [Upgrade your search service](#).

Concepts: search units, replicas, partitions

Capacity is expressed in *search units* that can be allocated in combinations of *partitions* and *replicas*.

 Expand table

Concept	Definition
Search unit	A single increment of total available capacity. A minimum of one search unit is required to run the service. Depending on your pricing tier, the maximum ranges from one to 36 units. The number of search units equals the number of replicas multiplied by the number of

Concept	Definition
	partitions: $R \times P = SU$. Each service starts with one replica and one partition, which consumes one unit: $1 \times 1 = 1$. Adding a second replica consumes two units: $2 \times 1 = 2$. A search unit is also the billing unit for a search service.
<i>Replica</i>	Instances of the search service, used primarily to load balance query operations. Each replica hosts one copy of an index. If you allocate three replicas, you have three copies of an index available for servicing query requests.
<i>Partition</i>	Physical storage and I/O for read/write operations (for example, when rebuilding or refreshing an index). Each partition has a slice of the total index. If you allocate three partitions, your index is divided into thirds.

Review the [partitions and replicas table](#) for possible combinations that stay under the 36 unit limit.

When to add capacity

Initially, a service is allocated a minimal level of resources consisting of one partition and one replica. The [tier you choose](#) determines partition size and speed, and each tier is optimized around a set of characteristics that fit various scenarios. If you choose a higher-end tier, you might [need fewer partitions](#) than if you go with S1. One of the questions you need to answer through self-directed testing is whether a larger and more expensive partition yields better performance than two cheaper partitions on a service provisioned at a lower tier.

A single service must have sufficient resources to handle all workloads (indexing and queries). Neither workload runs in the background. You can schedule indexing for times when query requests are naturally less frequent, but the service doesn't otherwise prioritize one task over another. Additionally, a certain amount of redundancy smooths out query performance when services or nodes are updated internally.

Guidelines for determining whether to add capacity include:

- Meeting the high availability criteria for service-level agreement.
- The frequency of HTTP 503 (Service unavailable) errors is increasing.
- The frequency of HTTP 429 (Too many requests) errors is increasing, an indication of low storage.
- Large query volumes are expected.
- A [one-time upgrade](#) to newer infrastructure and larger partitions isn't sufficient.
- The current number of partitions isn't adequate for indexing workloads.

As a general rule, search applications tend to need more replicas than partitions, particularly when the service operations are biased toward query workloads. Each replica is a copy of your

index, allowing the service to load balance requests against multiple copies. Azure AI Search manages all load balancing and replication of an index, and you can alter the number of replicas allocated for your service at any time. You can allocate up to 12 replicas in a Standard search service and 3 replicas in a Basic search service. Replica allocation can be made either from the [Azure portal](#) or one of the programmatic options.

Extra partitions are helpful for intensive indexing workloads. Extra partitions spread read/write operations across a larger number of compute resources.

Finally, larger indexes take longer to query. As such, you might find that every incremental increase in partitions requires a smaller but proportional increase in replicas. The complexity of your queries and query volume factors into how quickly query execution is turned around.

Note

Adding more replicas or partitions increases the cost of running the service, and can introduce slight variations in how results are ordered. Be sure to check the [pricing calculator](#) ↗ to understand the billing implications of adding more nodes. The [chart below](#) can help you cross-reference the number of search units required for a specific configuration. For more information on how extra replicas affect query processing, see [Ordering results](#).

How to upgrade capacity

Some Azure AI Search capabilities are only available to new services. One such capability is higher storage capacity, which applies to [services created after April 2024](#). However, if you created your service before April 2024, you can get higher capacity without recreating your service by performing a one-time upgrade. For more information, see [Upgrade your search service](#).

How to change capacity

To increase or decrease the capacity of your service, you have two options:

- [Add or remove partitions and replicas](#)
- [Change your pricing tier](#)

Add or remove partitions and replicas

1. Sign in to the [Azure portal](#) ↗ and select your search service.

2. From the left pane, select **Settings > Scale**.

The following screenshot shows a Standard service provisioned with one replica and partition. The formula at the bottom indicates how many search units are being used (1). If the unit price was \$100 (not a real price), the monthly cost of running this service would be \$100 on average.

Home > my-search-service

my-search-service | Scale

Search service

Search Save Discard Refresh Learn how to scale

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Resource visualizer Search management Settings Semantic ranker Knowledge Center Keys Scale Search traffic analytics Identity Networking Properties Locks Monitoring Automation Help

SEARCH UNITS 1/36

Pricing tier

Standard 25 GB/Partition, max 12 replicas, max 12 partitions, max 36 search units Change Pricing Tier

Replicas

Replicas distribute workloads across the service.

Partitions

Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple units.

1 Replicas x 1 Partitions = 1 of 36 available search units

3. Use the slider to increase or decrease the number of partitions, and then select **Save**.

This example adds a second replica and partition. Notice the search unit count; it's now four because the billing formula is replicas multiplied by partitions (2×2). Doubling capacity more than doubles the cost of running the service. If the search unit cost was \$100, the new monthly bill would now be \$400.

For the current per unit costs of each tier, visit the [pricing page](#).

Home > my-search-service

my-search-service | Scale ...

Search service

Save Discard Refresh Learn how to scale

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Resource visualizer Search management Settings Semantic ranker Knowledge Center Keys Scale Search traffic analytics Identity Networking Properties Locks Monitoring Automation

SEARCH UNITS 4/36

Pricing tier Pricing tier determines the physical storage of each partition, the request throughput of each additional replica, and the maximum number of indexes and other objects allowed in the service. [Learn more about pricing tiers for Azure AI Search](#)

Standard 25 GB/Partition, max 12 replicas, max 12 partitions, max 36 search units [Change Pricing Tier](#)

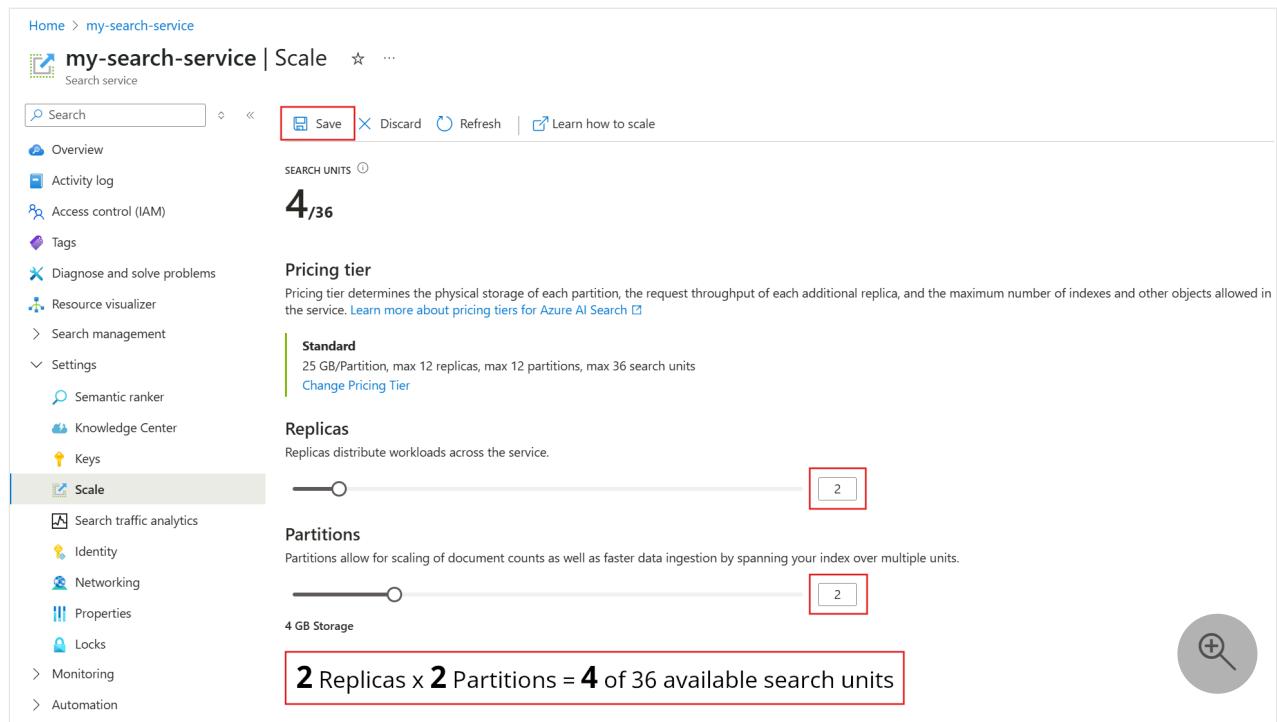
Replicas Replicas distribute workloads across the service.

Partitions Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple units.

4 GB Storage

2 Replicas x 2 Partitions = 4 of 36 available search units

🔍



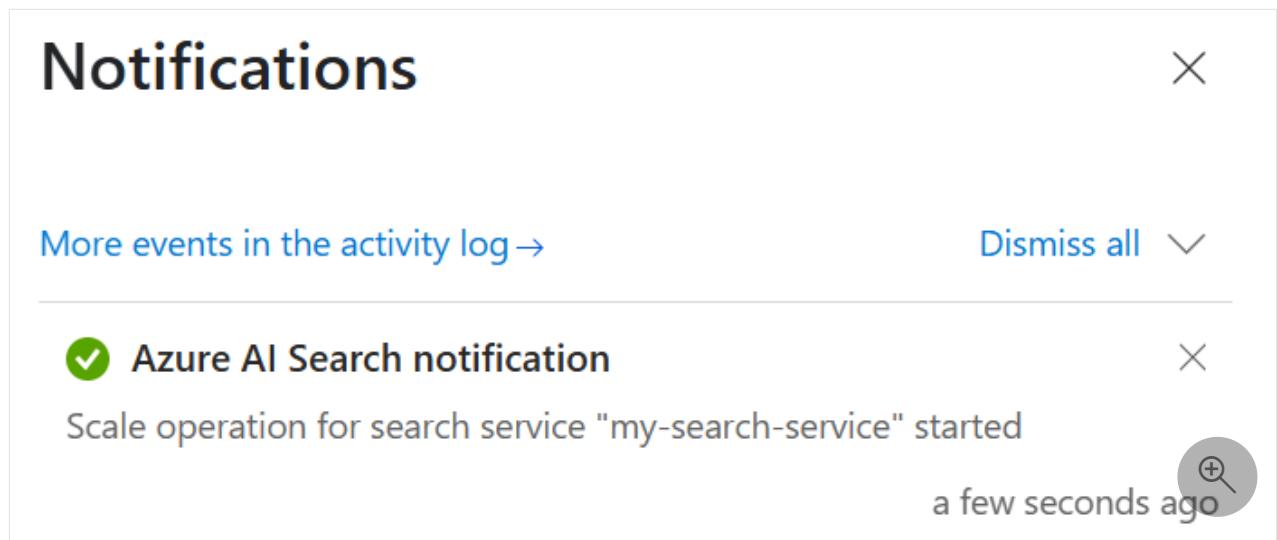
4. Check your notifications to confirm that the operation started.

Notifications

More events in the activity log → Dismiss all ×

✓ Azure AI Search notification X

Scale operation for search service "my-search-service" started a few seconds ago 🔍



This operation can take several hours to complete. It occurs in the background, so your search service remains fully operational and available for read and write operations.

You can't cancel the operation or monitor its progress. However, the following message displays while changes are underway:

Home > my-search-service

my-search-service | Scale ...

Search service

Save Discard Refresh Learn how to scale

Overview Activity log Access control (IAM)

Updating... This operation may take a while... 🔍



Change your pricing tier

! Note

The Azure portal and [Services - Update \(REST API\)](#) support changes between Basic and Standard (S1, S2, and S3) tiers. You can upgrade or downgrade tiers, provided your current service configuration doesn't exceed the [limits of the target tier](#). Your region also can't have [capacity constraints on the target tier](#).

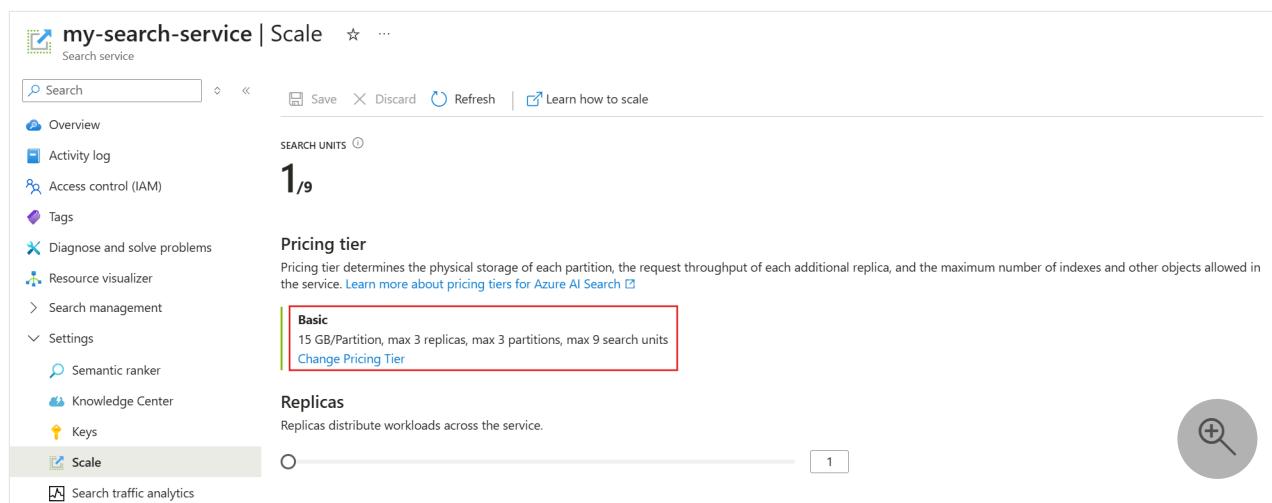
Your [pricing tier](#) determines the maximum storage of your search service. If you need more or less capacity, you can switch to a different pricing tier that accommodates your storage needs.

In addition to capacity, pricing tiers determine limits on indexes, indexers, and other search objects. Compare the [service limits](#) of your current tier and your desired tier before you proceed. Generally, switching to a higher tier increases your [storage limit](#) and [vector limit](#), increases request throughput, and decreases latency, while switching to a lower tier has the opposite effect.

Switching to a higher pricing tier also increases the cost of running your search service. For more information, see the [pricing page](#).

To change your pricing tier:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Settings > Scale**.
3. Under your current tier, select **Change Pricing Tier**.



4. On the **Select Pricing Tier** page, choose a different tier from the list.

You can switch between Basic, S1, S2, and S3, but you can't switch to or from Free, S3HD, L1, or L2. These tiers aren't selectable and appear dimmed.

Home > my-search-service | Scale >

Select Pricing Tier

Browse available skus and their features

Sku	Offering	Indexes	Indexers	Vector quota	Total storage	Search units	Replicas	Partitio...	Search unit cost...
F	Free	3	3	25 MB	50 MB	1	1	1	\$0.00
B	Basic	15	15	1 GB/Partition	2 GB/Partition	3	3	1	\$0.00
S	Standard	50	50	3 GB/Partition	25 GB/Partition	36	12	12	\$0.00
S2	Standard	200	200	12 GB/Partition	100 GB/Partition	36	12	12	\$0.00
S3	Standard	200	200	36 GB/Partition	200 GB/Partition	36	12	12	\$0.00
S3HD	High-density	1000	0	36 GB/Partition	200 GB/Partition	36	12	3	\$0.00
L1	Storage Optimized	10	10	12 GB/Partition	1 TB/Partition	36	12	12	\$0.00
L2	Storage Optimized	10	10	36 GB/Partition	2 TB/Partition	36	12	12	\$0.00

Select Prices presented are estimates in your local currency that include only Azure infrastructure costs and any discounts for the subscription and location. The prices don't include any applicable software costs. Final charges will appear in your local currency in cost analysis and billing views. [View Azure pricing calculator.](#)



5. To start the scale operation, select Save.

my-search-service | Scale

SEARCH UNITS 1/36

Pricing tier Standard

160 GB/Partition, max 12 replicas, max 12 partitions, max 36 search units

[Change Pricing Tier](#)




This operation can take several hours to complete. It occurs in the background, so your search service remains fully operational and available for read and write operations.

You can't cancel the operation or monitor its progress. However, the following message displays while changes are underway:

my-search-service | Scale

Updating... This operation may take a while...



How scale requests are handled

Upon receipt of a scale request, the search service:

1. Checks whether the request is valid.

2. Starts backing up data and system information.
3. Checks whether the service is already in a provisioning state (currently adding or eliminating either replicas or partitions).
4. Starts provisioning.

Scaling a service can take as little as 15 minutes or well over an hour, depending on the size of the service and the scope of the request. Backup can take several minutes, depending on the amount of data and number of partitions and replicas.

The above steps aren't entirely consecutive. For example, the system starts provisioning when it can safely do so, which could be while backup is winding down.

Errors during scaling

The following table lists causes and solutions for errors that can occur during scaling operations.

[] Expand table

Error message	Cause	Solution
"Service update operations aren't allowed at this time because we're processing a previous request."	Another scaling operation is in progress.	Check the Overview page in the Azure portal or use the Search Management REST API , Azure PowerShell , or Azure CLI to get the status of your search service. If the status is "Provisioning," wait until it becomes "Succeeded" or "Failed" before you try again. ^{1, 2}
"Failed to scale search service <i>servicename</i> . Error: <i>Object count ActualCount</i> exceeds allowable limit: <i>MaximumCount</i> ."	Your current service configuration exceeds the limits of the target pricing tier.	Check that your storage usage, vector usage, indexes, indexers, and other objects fit within the lower tier's service limits . For example, the Basic tier supports up to 15 indexes, so you can't switch from S1 to Basic if you have 16 indexes. Adjust your resources before you try again.

¹ There's no status for backups, which are internal operations that are unlikely to disrupt a scaling exercise.

² If your search service appears to be stalled in a provisioning state, check for orphaned indexes that are unusable, with zero query volumes and no index updates. An unusable index can block changes to service capacity. In particular, look for [CMK-encrypted](#) indexes whose keys are no longer valid. Either delete the index or restore the keys to bring the index back online and unblock your scaling operation.

Partition and replica combinations

The following chart applies to Standard tier and higher. It shows all possible combinations of partitions and replicas, subject to the 36 search unit maximum per service.

[+] Expand table

	1 partition	2 partitions	3 partitions	4 partitions	6 partitions	12 partitions
1 replica	1 SU	2 SU	3 SU	4 SU	6 SU	12 SU
2 replicas	2 SU	4 SU	6 SU	8 SU	12 SU	24 SU
3 replicas	3 SU	6 SU	9 SU	12 SU	18 SU	36 SU
4 replicas	4 SU	8 SU	12 SU	16 SU	24 SU	N/A
5 replicas	5 SU	10 SU	15 SU	20 SU	30 SU	N/A
6 replicas	6 SU	12 SU	18 SU	24 SU	36 SU	N/A
12 replicas	12 SU	24 SU	36 SU	N/A	N/A	N/A

Basic search services have lower search unit counts.

- On search services created before April 3, 2024, Basic services can have exactly one partition and up to three replicas for a maximum limit of three SUs. The only adjustable resource is replicas. However, you might be able to increase your partition count by [upgrading your service](#).
- On search services created after April 3, 2024 in [supported regions](#), Basic services can have up to three partitions and three replicas. The maximum SU limit is nine to support a full complement of partitions and replicas.

For search services on any billable tier, regardless of creation date, you need a minimum of two replicas for high availability on queries.

For billing rates per tier and currency, see the [Azure AI Search pricing page](#).

Estimate capacity using a billable tier

The size of the indexes you expect to build determines storage needs. There are no solid heuristics or generalities that help with estimates. The only way to determine the size of an index is [build one](#). Its size is based on tokenization and embeddings, and whether you enable suggesters, filtering, and sorting, or can take advantage of [vector compression](#).

We recommend estimating on a billable tier, Basic or higher. The Free tier runs on physical resources shared by multiple customers and is subject to factors beyond your control. Only the dedicated resources of a billable search service can accommodate larger sampling and processing times for more realistic estimates of index quantity, size, and query volumes during development.

1. [Review service limits at each tier](#) to determine whether lower tiers can support the number of indexes you need. Consider whether you need multiple copies of an index for active development, testing, and production.

A search service is subject to object limits (maximum number of indexes, indexers, skillsets, etc.) and storage limits. Whichever limit is reached first is the effective limit.

2. [Create a service at a billable tier](#). Tiers are optimized for certain workloads. For example, the Storage Optimized tier has a limit of 10 indexes because it's designed to support a low number of large indexes.

- Start low, at Basic or S1, if you're not sure about the projected load.
- Start high, at S2 or even S3, if testing includes large-scale indexing and query loads.
- Start with Storage Optimized, at L1 or L2, if you're indexing a large amount of data and query load is relatively low, as with an internal business application.

3. [Build an initial index](#) to determine how source data translates to an index. This is the only way to estimate index size. Attributes on the field definitions affect physical storage requirements:

- For keyword search, marking fields as filterable and sortable [increases index size](#).
- For vector search, you can [set parameters to reduce vector size](#).

4. [Monitor storage, service limits, query volume, and latency](#) in the Azure portal. the Azure portal shows you queries per second, throttled queries, and search latency. All of these values can help you decide if you selected the right tier.

5. Add replicas for high availability or to mitigate slow query performance.

There are no guidelines on how many replicas are needed to accommodate query loads. Query performance depends on the complexity of the query and competing workloads. Although adding replicas clearly results in better performance, the result isn't strictly linear: adding three replicas doesn't guarantee triple throughput. For guidance in estimating QPS for your solution, see [Analyze performance](#)and [Monitor queries](#).

For an [inverted index](#), size and complexity are determined by content, not necessarily by the amount of data that you feed into it. A large data source with high redundancy could result in a smaller index than a smaller dataset that contains highly variable content. So it's rarely possible to infer index size based on the size of the original dataset.

Storage requirements can be inflated if you include data that will never be searched. Ideally, documents contain only the data that you need for the search experience.

Service-level agreement considerations

The Free tier and preview features aren't covered by [service-level agreements \(SLAs\)](#). For all billable tiers, SLAs take effect when you provision sufficient redundancy for your service.

- Two or more replicas satisfy query (read) SLAs.
- Three or more replicas satisfy query and indexing (read-write) SLAs.

The number of partitions doesn't affect SLAs.

Next steps

[Plan and manage costs](#)

Last updated on 11/19/2025

Plan and manage costs of an Azure AI Search service

This article explains how Azure AI Search is billed, including fixed and variable costs, and provides guidance for cost management.

Before you create a search service, use the [Azure pricing calculator](#) to estimate costs based on your planned [capacity](#) and features. Another resource is a capacity-planning worksheet that models your expected index size, indexing throughput, and indexing costs.

As your search workload evolves, follow our tips to minimize costs during both deployment and operation. You can also use built-in metrics to monitor query requests and [Cost Management](#) to create budgets, alerts, and data exports.

! Note

Higher-capacity partitions are available at the same billing rate on services created after April and May 2024. For more information about partition-size upgrades, see [Service limits](#).

Understand the billing model

Azure AI Search has both fixed and pay-as-you-go billing. You pay a fixed rate for your search service as long as it exists, while premium features are billed according to your usage.

Costs for Azure AI Search are only a portion of the monthly costs in your Azure bill. Although this article focuses on planning and managing Azure AI Search costs, you're billed for all Azure services and resources used in your Azure subscription, including non-Microsoft services.

How you're charged for the base service

When you create or use search resources, you're charged for the minimum required replica and partition combination ($R \times P$) at the prorated hourly rate of your [pricing tier](#). As your search units increase or decrease, so do your costs. For more information and an example of the billing model, see [Billing rates](#).

How you're charged for premium features

Premium features are charged in addition to the base cost of your search service. The following table lists premium features and their billing units. All of these features are optional, so if you

don't use them, you don't incur any charges.

 Expand table

Feature	Billing unit
Image extraction (AI enrichment) ¹	Per 1,000 images. See the pricing page .
Custom Entity Lookup skill (AI enrichment)	Per 1,000 text records. See the pricing page .
Built-in or custom skills (AI enrichment) ²	Number of transactions. Billed at the rate of the model provider: Foundry Tools, Azure OpenAI, or Microsoft Foundry.
Vectorizers ²	Number of vectorization operations. Billed at the rate of the model provider: Azure Vision in Foundry Tools, Azure OpenAI, or Foundry.
Semantic ranker	Number of queries of <code>queryType=semantic</code> . Billed at a progressive rate. See the pricing page .
Agentic retrieval	Number of agentic reasoning tokens, plus number of tokens used in query planning and answer formulation. See the pricing page .
Shared private link	Billed for bandwidth as long as the shared private link exists and is used.

¹ Refers to images extracted from a file within the indexer pipeline. Text extraction is free. Image extraction is billed when you [enable the indexAction parameter](#) or when you call the [Document Extraction skill](#).

² Charges for Azure OpenAI models and Foundry models appear on your bill for those services.

How you're otherwise charged

Depending on your configuration and usage, the following charges might apply:

- Data traffic might incur networking costs. See the [bandwidth pricing](#).
- Several premium features, such as [knowledge stores](#), [debug sessions](#), and [enrichment caches](#), depend on Azure Storage and incur storage costs. Charges for these features appear on your Azure Storage bill.
- [Customer-managed keys](#), which provide double encryption of sensitive content, require a billable [Azure Key Vault](#).
- A skillset can include [billable built-in skills](#), nonbillable built-in utility skills, and custom skills. Nonbillable utility skills include [Conditional](#), [Shaper](#), [Text Merge](#), and [Text Split](#). They

don't have an API key requirement or 20-document limit.

- A custom skill is functionality you provide. Custom skills are billable only if they call other billable services. They don't have an API key requirement or 20-document limit.

 **Note**

You aren't billed for the number of full-text or vector queries, query responses, or documents ingested. However, [service limits](#) apply to each pricing tier.

Estimate and plan costs

Use the [Azure pricing calculator](#) to estimate your baseline costs for Azure AI Search. You can also find estimated costs and tier comparisons on the [Select Pricing Tier](#) page during service creation.

For initial testing, we recommend that you create a capacity-planning worksheet. The worksheet helps you understand the index-to-source ratio and the effect of enrichment or vector features on both capacity and cost.

To create a capacity-planning worksheet:

1. Index a small sample (1–5%) of your data. Include any [OCR](#), enrichment, or embedding skills you plan to use.
2. Measure the index size, indexing throughput, and indexing costs.
3. Extrapolate the results to estimate the full-scale requirements for your data.

Minimize costs

To minimize the costs of your Azure AI Search solution, use the following strategies:

Deployment and configuration

- Create a search service in a [region with more storage per partition](#).
- Create all related Azure resources in the same region (or as few regions as possible) to minimize or eliminate bandwidth charges.
- Choose the lightest [pricing tier](#) that meets your needs. Basic and S1 offer full access to the modern API at the lowest hourly rate per SU.

- Use [Azure Web Apps](#) for your front-end application to keep requests and responses within the data center boundary.

Scaling

- Add [partitions](#) only when the index size or ingestion throughput requires it.
- Add [replicas](#) only when your queries per second increase, when complex queries are throttling your service, or when high availability is required.
- Scale up for resource-intensive operations, such as indexing, and then readjust downwards for regular query workloads.
- Write code to automate scaling for predictable workload patterns.
- Remember that capacity and pricing aren't linear. Doubling capacity more than doubles costs on the same tier. For better performance at a similar price, consider [switching to a higher tier](#).

Indexing and enrichment

- Use [incremental indexing](#) to process only new or changed data.
- Use [enrichment caching](#) and a [knowledge store](#) to reuse previously enriched content. Although caching incurs a storage charge, it lowers the cumulative cost of [AI enrichment](#).
- Keep vector payloads compact. For vector search, see the [vector compression best practices](#).

Monitor costs

At the service level, you can [monitor built-in metrics](#) for your queries per second (QPS), search latency, throttled queries, and index size. You can then [create an Azure Monitor dashboard](#) that overlays QPS, latency, and cost data to determine when to add or remove replicas.

At the subscription or resource group level, [Cost Management](#) provides tools to track, analyze, and control costs. You can use Cost Management to:

- [Create budgets](#) that define and track progress against spending limits. For more granular monitoring, customize your budgets using [filters](#) for specific Azure resources or services. Filters prevent you from accidentally creating resources that incur extra costs.

- [Create alerts](#) that automatically notify stakeholders of spending anomalies or overspending risks. Alerts are based on spending compared to budget and cost thresholds. Both budgets and alerts are created for subscriptions and resource groups, making them useful for monitoring overall costs.
- [Export cost data](#) to a storage account. This is helpful when you or others need to perform more cost analysis. For example, a finance team can analyze the data using Excel or Power BI. You can export your costs on a daily, weekly, or monthly schedule and set a custom date range. Exporting cost data is the recommended method for retrieving cost datasets.

FAQ

Can I temporarily shut down a search service to save on costs?

Search runs as a continuous service. Dedicated resources are always operational and allocated for your exclusive use for the lifetime of your service. To stop billing entirely, you must delete the service. Deleting a service is permanent and also deletes its associated data.

Can I change the billing rate (tier) of an existing search service?

Existing services can switch between Basic and Standard (S1, S2, and S3) tiers. Your current service configuration can't exceed the limits of the target tier, and your region can't have capacity constraints on the target tier. For more information, see [Change your pricing tier](#).

Related content

- [Azure AI Search pricing ↗](#)
- [Choose a pricing tier for Azure AI Search](#)
- [Optimize your cloud investment with Cost Management](#)
- [Quickstart: Start using Cost analysis](#)

Last updated on 11/10/2025

Design patterns for multitenant SaaS applications and Azure AI Search

05/29/2025

A multitenant application is one that provides the same services and capabilities to any number of tenants who can't see or share the data of any other tenant. This article discusses tenant isolation strategies for multitenant applications built with Azure AI Search.

Azure AI Search concepts

As a search-as-a-service solution, [Azure AI Search](#) allows developers to add rich search experiences to applications without managing any infrastructure or becoming an expert in information retrieval. Data is uploaded to the service and then stored in the cloud. Using simple requests to the Azure AI Search API, the data can then be modified and searched.

Search services, indexes, fields, and documents

Before discussing design patterns, it's important to understand a few basic concepts.

When using Azure AI Search, one subscribes to a *search service*. As data is uploaded to Azure AI Search, it's stored in an *index* within the search service. There can be a number of indexes within a single service. To use the familiar concepts of databases, the search service can be likened to a database while the indexes within a service can be likened to tables within a database.

Each index within a search service has its own schema, which is defined by a number of customizable *fields*. Data is added to an Azure AI Search index in the form of individual *documents*. Each document must be uploaded to a particular index and must fit that index's schema. When searching data using Azure AI Search, the full-text search queries are issued against a particular index. To compare these concepts to those of a database, fields can be likened to columns in a table and documents can be likened to rows.

Scalability

Any Azure AI Search service in the Standard [pricing tier](#) can scale in two dimensions: storage and availability.

- *Partitions* can be added to increase the storage of a search service.
- *Replicas* can be added to a service to increase the throughput of requests that a search service can handle.

Adding and removing partitions and replicas at will allow the capacity of the search service to grow with the amount of data and traffic the application demands. In order for a search service to achieve a read [SLA](#), it requires two replicas. In order for a service to achieve a read-write [SLA](#), it requires three replicas.

Service and index limits in Azure AI Search

There are a few different [pricing tiers](#) in Azure AI Search, each of the tiers has different [limits and quotas](#). Some of these limits are at the service-level, some are at the index-level, and some are at the partition-level.

S3 High Density

In Azure AI Search's S3 pricing tier, there's an option for the High Density (HD) mode designed specifically for multitenant scenarios. In many cases, it's necessary to support a large number of smaller tenants under a single service to achieve the benefits of simplicity and cost efficiency.

S3 HD allows for the many small indexes to be packed under the management of a single search service by trading the ability to scale out indexes using partitions for the ability to host more indexes in a single service.

An S3 service is designed to host a fixed number of indexes (maximum 200) and allow each index to scale in size horizontally as new partitions are added to the service. Adding partitions to S3 HD services increases the maximum number of indexes that the service can host. The ideal maximum size for an individual S3HD index is around 50 - 80 GB, although there's no hard size limit on each index imposed by the system.

Considerations for multitenant applications

Multitenant applications must effectively distribute resources among the tenants while preserving some level of privacy between the various tenants. There are a few considerations when designing the architecture for such an application:

- *Tenant isolation:* Application developers need to take appropriate measures to ensure that no tenants have unauthorized or unwanted access to the data of other tenants. Beyond the perspective of data privacy, tenant isolation strategies require effective management of shared resources and protection from noisy neighbors.
- *Cloud resource cost:* As with any other application, software solutions must remain cost competitive as a component of a multitenant application.

- *Ease of Operations*: When developing a multitenant architecture, the impact on the application's operations and complexity is an important consideration. Azure AI Search has a [99.9% SLA](#).
- *Global footprint*: Multitenant applications often need to serve tenants who are distributed across the globe.
- *Scalability*: Application developers need to consider how they reconcile between maintaining a sufficiently low level of application complexity and designing the application to scale with number of tenants and the size of tenants' data and workload.

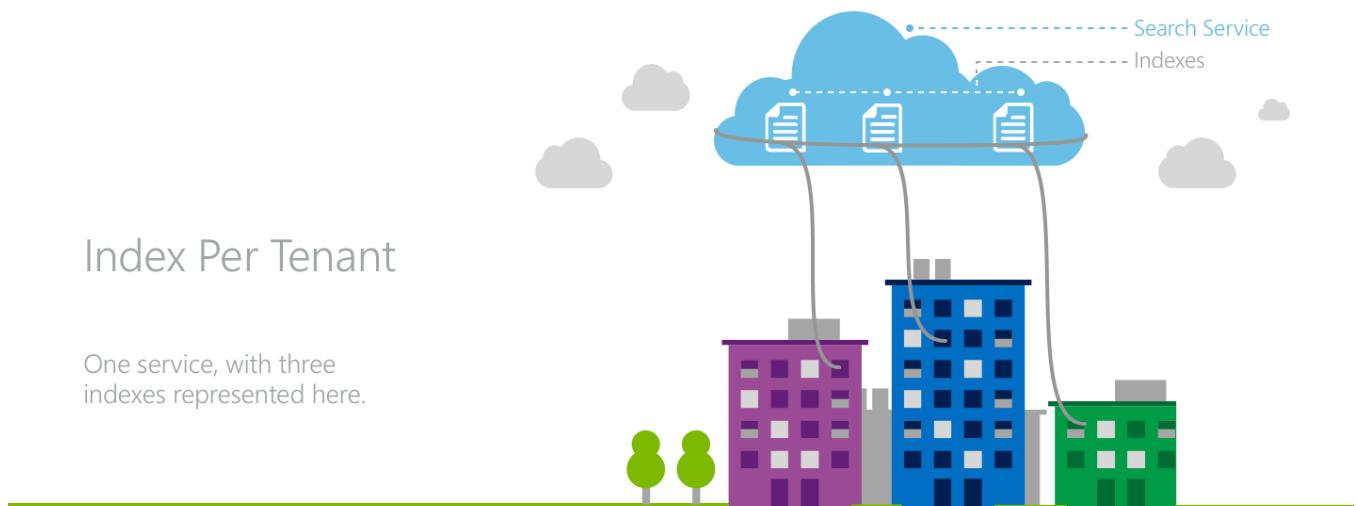
Azure AI Search offers a few boundaries that can be used to isolate tenants' data and workload.

Modeling multitenancy with Azure AI Search

In the case of a multitenant scenario, the application developer consumes one or more search services and divides their tenants among services, indexes, or both. Azure AI Search has a few common patterns when modeling a multitenant scenario:

- *One index per tenant*: Each tenant has its own index within a search service that is shared with other tenants.
- *One service per tenant*: Each tenant has its own dedicated Azure AI Search service, offering highest level of data and workload separation.
- *Mix of both*: Larger, more-active tenants are assigned dedicated services while smaller tenants are assigned individual indexes within shared services.

Model 1: One index per tenant



In an index-per-tenant model, multiple tenants occupy a single Azure AI Search service where each tenant has their own index.

Tenants achieve data isolation because all search requests and document operations are issued at an index level in Azure AI Search. In the application layer, there's the need awareness to direct the various tenants' traffic to the proper indexes while also managing resources at the service level across all tenants.

A key attribute of the index-per-tenant model is the ability for the application developer to oversubscribe the capacity of a search service among the application's tenants. If the tenants have an uneven distribution of workload, the optimal combination of tenants can be distributed across a search service's indexes to accommodate a number of highly active, resource-intensive tenants while simultaneously serving a long tail of less active tenants. The trade-off is the inability of the model to handle situations where each tenant is concurrently highly active.

The index-per-tenant model provides the basis for a variable cost model, where an entire Azure AI Search service is bought up-front and then subsequently filled with tenants. This allows for unused capacity to be designated for trials and free accounts.

For applications with a global footprint, the index-per-tenant model might not be the most efficient. If an application's tenants are distributed across the globe, a separate service can be necessary for each region, duplicating costs across each of them.

Azure AI Search allows for the scale of both the individual indexes and the total number of indexes to grow. If an appropriate pricing tier is chosen, partitions and replicas can be added to the entire search service when an individual index within the service grows too large in terms of storage or traffic.

If the total number of indexes grows too large for a single service, another service has to be provisioned to accommodate the new tenants. If indexes have to be moved between search services as new services are added, the data from the index has to be manually copied from one index to the other as Azure AI Search doesn't allow for an index to be moved.

Model 2: One service per tenant



In a service-per-tenant architecture, each tenant has its own search service.

In this model, the application achieves the maximum level of isolation for its tenants. Each service has dedicated storage and throughput for handling search requests. Each tenant has individual ownership of API keys.

For applications where each tenant has a large footprint or the workload has little variability from tenant to tenant, the service-per-tenant model is an effective choice as resources aren't shared across various tenants' workloads.

A service per tenant model also offers the benefit of a predictable, fixed cost model. There's no up-front investment in an entire search service until there's a tenant to fill it, however the cost-per-tenant is higher than an index-per-tenant model.

The service-per-tenant model is an efficient choice for applications with a global footprint. With geographically distributed tenants, it's easy to have each tenant's service in the appropriate region.

The challenges in scaling this pattern arise when individual tenants outgrow their service. Azure AI Search doesn't currently support upgrading the pricing tier of a search service, so all data would have to be manually copied to a new service.

Model 3: Hybrid

Another pattern for modeling multitenancy is mixing both index-per-tenant and service-per-tenant strategies.

By mixing the two patterns, an application's largest tenants can occupy dedicated services while the long tail of less active, smaller tenants can occupy indexes in a shared service. This

model ensures that the largest tenants have consistently high performance from the service while helping to protect the smaller tenants from any noisy neighbors.

However, implementing this strategy relies on foresight in predicting which tenants will require a dedicated service versus an index in a shared service. Application complexity increases with the need to manage both of these multitenancy models.

Achieving even finer granularity

The above design patterns to model multitenant scenarios in Azure AI Search assume a uniform scope where each tenant is a whole instance of an application. However, applications can sometimes handle many smaller scopes.

If service-per-tenant and index-per-tenant models aren't sufficiently small scopes, it's possible to model an index to achieve an even finer degree of granularity.

To have a single index behave differently for different client endpoints, a field can be added to an index, which designates a certain value for each possible client. Each time a client calls Azure AI Search to query or modify an index, the code from the client application specifies the appropriate value for that field using Azure AI Search's [filter](#) capability at query time.

This method can be used to achieve functionality of separate user accounts, separate permission levels, and even completely separate applications.

Note

Using the approach described above to configure a single index to serve multiple tenants affects the relevance of search results. Search relevance scores are computed at an index-level scope, not a tenant-level scope, so all tenants' data is incorporated in the relevance scores' underlying statistics such as term frequency.

Next steps

Azure AI Search is a compelling choice for many applications. When evaluating the various design patterns for multitenant applications, consider the [various pricing tiers](#) and the respective [service limits](#) to best tailor Azure AI Search to fit application workloads and architectures of all sizes.

Manage your Azure AI Search service using PowerShell

08/01/2025

You can run PowerShell cmdlets and scripts on Windows, Linux, or in Azure Cloud Shell to create and configure Azure AI Search.

Use the [Az.Search module](#) to perform the following tasks:

- ✓ List search services in a subscription
- ✓ Return service information
- ✓ Create or delete a service
- ✓ Create a service with a private endpoint
- ✓ Regenerate admin API-keys
- ✓ Create or delete query api-keys
- ✓ Scale up or down with replicas and partitions
- ✓ Create a shared private link resource

Occasionally, questions are asked about tasks *not* on the above list.

You can't change the name or region of a service programmatically or in the Azure portal. Dedicated resources are allocated when a service is created, so changing the underlying hardware (location or node type) requires a new service.

ⓘ Note

The [Search Management REST APIs](#) and [Azure portal](#) support changing your pricing tier. Currently, you can only switch between Basic and Standard (S1, S2, and S3) tiers.

You can't use tools or APIs to transfer content, such as an index, from one service to another. Within a service, programmatic creation of content is through [Search Service REST API](#) or an SDK such as [Azure SDK for .NET](#). While there are no dedicated commands for content migration, you can write script that calls REST API or a client library to create and load indexes on a new service.

Preview administration features are typically not available in the [Az.Search module](#). If you want to use a preview feature, [use the Management REST API](#) and a preview API version.

The [Az.Search module](#) extends [Azure PowerShell](#) with full parity to the stable versions of the [Search Management REST APIs](#).

Check versions and load modules

The examples in this article are interactive and require elevated permissions. Local PowerShell and the Azure PowerShell (the Az module) are required.

PowerShell version check

Install the latest version of PowerShell if you don't have it.

```
Azure PowerShell
```

```
$PSVersionTable.PSVersion
```

Load Azure PowerShell

If you aren't sure whether Az is installed, run the following command as a verification step.

```
Azure PowerShell
```

```
Get-InstalledModule -Name Az
```

Some systems don't autoload modules. If you got an error on the previous command, try loading the module, and if that fails, go back to the installation [Azure PowerShell installation instructions](#) to see if you missed a step.

```
Azure PowerShell
```

```
Import-Module -Name Az
```

Connect to Azure with a browser sign-in token

You can use your portal sign-in credentials to connect to a subscription in PowerShell.

Alternatively you can [authenticate non-interactively with a service principal](#).

```
Azure PowerShell
```

```
Connect-AzAccount
```

If you hold multiple Azure subscriptions, set your Azure subscription. To see a list of your current subscriptions, run this command.

```
Azure PowerShell
```

```
Get-AzSubscription | sort SubscriptionName | Select SubscriptionName
```

To specify the subscription, run the following command. In the following example, the subscription name is `ContosoSubscription`.

```
Azure PowerShell
```

```
Select-AzSubscription -SubscriptionName ContosoSubscription
```

List services in a subscription

The following commands are from [Az.Resources](#), returning information about existing resources and services already provisioned in your subscription. If you don't know how many search services are already created, these commands return that information, saving you a trip to the Azure portal.

The first command returns all search services.

```
Azure PowerShell
```

```
Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
```

From the list of services, return information about a specific resource.

```
Azure PowerShell
```

```
Get-AzResource -ResourceName <service-name>
```

Results should look similar to the following output.

```
Name : my-demo-searchapp
ResourceGroupName : demo-westus
ResourceType : Microsoft.Search/searchServices
Location : westus
ResourceId : /subscriptions/<alphanumeric-subscription-ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
```

Import Az.Search

Commands from [Az.Search](#) aren't available until you load the module.

```
Azure PowerShell
```

```
Install-Module -Name Az.Search -Scope CurrentUser
```

List all Az.Search commands

As a verification step, return a list of commands provided in the module.

```
Azure PowerShell
```

```
Get-Command -Module Az.Search
```

Results should look similar to the following output.

CommandType	Name	Version
Source		
-----	----	-----
---		---
Cmdlet	Get-AzSearchAdminKeyPair	0.10.0
Az.Search	Get-AzSearchPrivateEndpointConnection	0.10.0
Cmdlet	Get-AzSearchPrivateLinkResource	0.10.0
Az.Search	Get-AzSearchQueryKey	0.10.0
Cmdlet	Get-AzSearchService	0.10.0
Az.Search	Get-AzSearchSharedPrivateLinkResource	0.10.0
Cmdlet	New-AzSearchAdminKey	0.10.0
Az.Search	New-AzSearchQueryKey	0.10.0
Cmdlet	New-AzSearchService	0.10.0
Az.Search	New-AzSearchSharedPrivateLinkResource	0.10.0
Cmdlet	Remove-AzSearchPrivateEndpointConnection	0.10.0
Az.Search	Remove-AzSearchQueryKey	0.10.0
Cmdlet	Remove-AzSearchService	0.10.0

Az.Search		
Cmdlet	Remove-AzSearchSharedPrivateLinkResource	0.10.0
Az.Search		
Cmdlet	Set-AzSearchPrivateEndpointConnection	0.10.0
Az.Search		
Cmdlet	Set-AzSearchService	0.10.0
Az.Search		
Cmdlet	Set-AzSearchSharedPrivateLinkResource	0.10.0
Az.Search		

If you have an older version of the package, update the module to get the latest functionality.

Azure PowerShell

```
Update-Module -Name Az.Search
```

Get search service information

After **Az.Search** is imported and you know the resource group containing your search service, run [Get-AzSearchService](#) to return the service definition, including name, region, tier, and replica and partition counts. For this command, provide the resource group that contains the search service.

Azure PowerShell

```
Get-AzSearchService -ResourceGroupName <resource-group-name>
```

Results should look similar to the following output.

```
Name          : my-demo-searchapp
ResourceGroupName : demo-westus
ResourceType    : Microsoft.Search/searchServices
Location       : West US
Sku           : Standard
ReplicaCount   : 1
PartitionCount : 1
HostingMode    : Default
ResourceId     : /subscriptions/<alphanumeric-subscription-ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
```

Create or delete a service

New-AzSearchService is used to [create a new search service](#).

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> -Name <search-service-name> -Sku "Standard" -Location "West US" -PartitionCount 3 -ReplicaCount 3 -HostingMode Default
```

Results should look similar to the following output.

Azure PowerShell

```
ResourceGroupName : demo-westus
Name             : my-demo-searchapp
Id               : /subscriptions/<alphanumeric-subscription-ID>/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
Location         : West US
Sku              : Standard
ReplicaCount     : 3
PartitionCount   : 3
HostingMode      : Default
Tags
```

Remove-AzSearchService is used to delete a service and its data.

Azure PowerShell

```
Remove-AzSearchService -ResourceGroupName <resource-group-name> -Name <search-service-name>
```

You're asked to confirm the action.

Azure PowerShell

```
Confirm
Are you sure you want to remove Search Service 'pstestazuresearch01'?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

Create a service with IP rules

Depending on your security requirements, you might want to create a search service with an [IP firewall configured](#). To do so, first define the IP Rules and then pass them to the `IPRuleList` parameter as shown below.

Azure PowerShell

```
$ipRules = @([pscustomobject]@{Value="55.5.63.73"},  
            [pscustomobject]@{Value="52.228.215.197"},  
            [pscustomobject]@{Value="101.37.221.205"})  
  
New-AzSearchService -ResourceGroupName <resource-group-name> `  
                    -Name <search-service-name> `  
                    -Sku Standard `  
                    -Location "West US" `  
                    -PartitionCount 3 -ReplicaCount 3 `  
                    -HostingMode Default `  
                    -IPRuleList $ipRules
```

Create a service with a system assigned managed identity

In some cases, such as when [using managed identity to connect to a data source](#), you need to turn on [system assigned managed identity](#). This is done by adding `-IdentityType SystemAssigned` to the command.

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> `  
                    -Name <search-service-name> `  
                    -Sku Standard `  
                    -Location "West US" `  
                    -PartitionCount 3 -ReplicaCount 3 `  
                    -HostingMode Default `  
                    -IdentityType SystemAssigned
```

Create an S3HD service

To create an [S3HD](#) service, a combination of `-Sku` and `-HostingMode` is used. Set `-Sku` to `Standard3` and `-HostingMode` to `HighDensity`.

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> `  
                    -Name <search-service-name> `  
                    -Sku Standard3 `  
                    -Location "West US" `  
                    -PartitionCount 1 -ReplicaCount 3 `  
                    -HostingMode HighDensity
```

Create a service with a private endpoint

[Private Endpoints](#) for Azure AI Search allow a client on a virtual network to securely access data in a search index over a [Private Link](#). The private endpoint uses an IP address from the [virtual network address space](#) for your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. For more information, see [Creating a private endpoint for Azure AI Search](#).

The following example shows how to create a search service with a private endpoint.

First, deploy a search service with `PublicNetworkAccess` set to `Disabled`.

Azure PowerShell

```
$searchService = New-AzSearchService`  
    -ResourceGroupName <search-service-resource-group-name>`  
    -Name <search-service-name>`  
    -Sku Standard`  
    -Location "West US" `  
    -PartitionCount 1 -ReplicaCount 1`  
    -HostingMode Default`  
    -PublicNetworkAccess Disabled
```

Next, create a virtual network, private network connection, and the private endpoint.

Azure PowerShell

```
# Create the subnet  
$subnetConfig = New-AzVirtualNetworkSubnetConfig`  
    -Name <subnet-name>`  
    -AddressPrefix 10.1.0.0/24`  
    -PrivateEndpointNetworkPolicies Disabled`  
  
# Create the virtual network  
$virtualNetwork = New-AzVirtualNetwork`  
    -ResourceGroupName <vm-resource-group-name>`  
    -Location "West US" `  
    -Name <virtual-network-name>`  
    -AddressPrefix 10.1.0.0/16`  
    -Subnet $subnetConfig`  
  
# Create the private network connection  
$privateLinkConnection = New-AzPrivateLinkServiceConnection`  
    -Name <private-link-name>`  
    -PrivateLinkServiceId $searchService.Id`  
    -GroupId searchService`  
  
# Create the private endpoint  
$privateEndpoint = New-AzPrivateEndpoint`  
    -Name <private-endpoint-name>`  
    -ResourceGroupName <private-endpoint-resource-group-name>`
```

```
-Location "West US"
-Subnet $virtualNetwork.subnets[0]
-PrivateLinkServiceConnection $privateLinkConnection
```

Finally, create a private DNS Zone.

Azure PowerShell

```
## Create private dns zone
$zone = New-AzPrivateDnsZone
    -ResourceGroupName <private-dns-resource-group-name>
    -Name "privatelink.search.windows.net"

## Create dns network link
$link = New-AzPrivateDnsVirtualNetworkLink
    -ResourceGroupName <private-dns-link-resource-group-name>
    -ZoneName "privatelink.search.windows.net"
    -Name "myLink"
    -VirtualNetworkId $virtualNetwork.Id

## Create DNS configuration
$config = New-AzPrivateDnsZoneConfig
    -Name "privatelink.search.windows.net"
    -PrivateDnsZoneId $zone.ResourceId

## Create DNS zone group
New-AzPrivateDnsZoneGroup
    -ResourceGroupName <private-dns-zone-resource-group-name>
    -PrivateEndpointName <private-endpoint-name>
    -Name 'myZoneGroup'
    -PrivateDnsZoneConfig $config
```

For more information on creating private endpoints in PowerShell, see this [Private Link Quickstart](#).

Manage private endpoint connections

In addition to creating a private endpoint connection, you can also [Get](#), [Set](#), and [Remove](#) the connection.

[Get-AzSearchPrivateEndpointConnection](#) is used to retrieve a private endpoint connection and to see its status.

Azure PowerShell

```
Get-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name>
```

[Set-AzSearchPrivateEndpointConnection](#) is used to update the connection. The following example sets a private endpoint connection to rejected:

```
Azure PowerShell
```

```
Set-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <pe-connection-name> -Status Rejected -Description "Rejected"
```

[Remove-AzSearchPrivateEndpointConnection](#) is used to delete the private endpoint connection.

```
Azure PowerShell
```

```
Remove-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <pe-connection-name>
```

Regenerate admin keys

[New-AzSearchAdminKey](#) is used to roll over admin [API keys](#). Two admin keys are created with each service for authenticated access. Keys are required on every request. Both admin keys are functionally equivalent, granting full write access to a search service with the ability to retrieve any information, or create and delete any object. Two keys exist so that you can use one while replacing the other.

You can only regenerate one at a time, specified as either the `primary` or `secondary` key. For uninterrupted service, remember to update all client code to use a secondary key while rolling over the primary key. Avoid changing the keys while operations are in flight.

As you might expect, if you regenerate keys without updating client code, requests using the old key will fail. Regenerating all new keys doesn't permanently lock you out of your service, and you can still access the service through the Azure portal. After you regenerate primary and secondary keys, you can update client code to use the new keys and operations will resume accordingly.

Values for the API keys are generated by the service. You can't provide a custom key for Azure AI Search to use. Similarly, there's no user-defined name for admin API-keys. References to the key are fixed strings, either `primary` or `secondary`.

```
Azure PowerShell
```

```
New-AzSearchAdminKey -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -KeyKind Primary
```

Results should look similar to the following output. Both keys are returned even though you only change one at a time.

Primary	Secondary
-----	-----
<alphanumeric-guid>	<alphanumeric-guid>

Create or delete query keys

[New-AzSearchQueryKey](#) is used to create query [API keys](#) for read-only access from client apps to an Azure AI Search index. Query keys are used to authenticate to a specific index for retrieving search results. Query keys don't grant read-only access to other items on the service, such as an index, data source, or indexer.

You can't provide a key for Azure AI Search to use. API keys are generated by the service.

Azure PowerShell

```
New-AzSearchQueryKey -ResourceGroupName <search-service-resource-group-name> -  
ServiceName <search-service-name> -Name <query-key-name>
```

Scale replicas and partitions

[Set-AzSearchService](#) is used to [increase or decrease replicas and partitions](#) to readjust billable resources within your service. Increasing replicas or partitions adds to your bill, which has both fixed and variable charges. If you have a temporary need for more processing power, you can increase replicas and partitions to handle the workload. The monitoring area in the Overview portal page has tiles on query latency, queries per second, and throttling, indicating whether current capacity is adequate.

It can take a while to add or remove resourcing. Adjustments to capacity occur in the background, allowing existing workloads to continue. Extra capacity is used for incoming requests as soon as it's ready, with no extra configuration required.

Removing capacity can be disruptive. Stopping all indexing and indexer jobs prior to reducing capacity is recommended to avoid dropped requests. If that isn't feasible, you might consider reducing capacity incrementally, one replica and partition at a time, until your new target levels are reached.

Once you submit the command, there's no way to terminate it midway through. You have to wait until the command is finished before revising the counts.

Azure PowerShell

```
Set-AzSearchService -ResourceGroupName <search-service-resource-group-name> -Name <search-service-name> -PartitionCount 6 -ReplicaCount 6
```

Results should look similar to the following output.

```
ResourceGroupName : demo-westus
Name             : my-demo-searchapp
Location         : West US
Sku              : Standard
ReplicaCount     : 6
PartitionCount   : 6
HostingMode      : Default
Id               : /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-
eeeeee4e4e/resourceGroups/demo-
westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
```

Create a shared private link resource

Private endpoints of secured resources that are created through Azure AI Search APIs are referred to as *shared private link resources*. This is because you're "sharing" access to a resource, such as a storage account that has been integrated with the [Azure Private Link service](#).

If you're using an indexer to index data in Azure AI Search, and your data source is on a private network, you can create an outbound [private endpoint connection](#) to reach the data.

A full list of the Azure Resources for which you can create outbound private endpoints from Azure AI Search can be found [here](#) along with the related **Group ID** values.

[New-AzSearchSharedPrivateLinkResource](#) is used to create the shared private link resource. Keep in mind that some configuration might be required for the data source before running this command.

Azure PowerShell

```
New-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-service-resource-
group-name> -ServiceName <search-service-name> -Name <spl-name> -
PrivateLinkResourceId /subscriptions/<alphanumeric-subscription-
ID>/resourceGroups/<storage-resource-group-
```

```
name>/providers/Microsoft.Storage/storageAccounts/myBlobStorage -GroupId <group-id> -RequestMessage "Please approve"
```

[Get-AzSearchSharedPrivateLinkResource](#) allows you to retrieve the shared private link resources and view their status.

Azure PowerShell

```
Get-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <spl-name>
```

You need to approve the connection with the following command before it can be used.

Azure PowerShell

```
Approve-AzPrivateEndpointConnection  
-Name <spl-name>  
-ServiceName <search-service-name>  
-ResourceGroupName <search-service-resource-group-name>  
-Description = "Approved"
```

[Remove-AzSearchSharedPrivateLinkResource](#) is used to delete the shared private link resource.

Azure PowerShell

```
$job = Remove-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <spl-name> -Force -AsJob  
  
$job | Get-Job
```

For full details on setting up shared private link resources, see the documentation on [making indexer connections through a private endpoint](#).

Next steps

Build an [index](#), [query an index](#) using the Azure portal, REST APIs, or the .NET SDK.

- [Create an Azure AI Search index in the Azure portal](#)
- [Set up an indexer to load data from other services](#)
- [Query an Azure AI Search index using Search explorer in the Azure portal](#)
- [How to use Azure AI Search in .NET](#)

Manage your Azure AI Search service using the Azure CLI

09/30/2025

You can run Azure CLI commands and scripts on Windows, macOS, Linux, or in Azure Cloud Shell to create and configure Azure AI Search.

Use the [az search module](#) to perform the following tasks:

- ✓ List search services in a subscription
- ✓ Return service information
- ✓ Create or delete a service
- ✓ Create a service with a private endpoint
- ✓ Create a service with confidential computing
- ✓ Regenerate admin API-keys
- ✓ Create or delete query api-keys
- ✓ Scale up or down with replicas and partitions
- ✓ Create a shared private link resource

Occasionally, questions are asked about tasks *not* on the above list.

You can't change the name or region of a service programmatically or in the Azure portal. Dedicated resources are allocated when a service is created, so changing the underlying hardware (location or node type) requires a new service.

⚠ Note

The [Search Management REST APIs](#) and [Azure portal](#) support changing your pricing tier. Currently, you can only switch between Basic and Standard (S1, S2, and S3) tiers.

You can't use tools or APIs to transfer content, such as an index, from one service to another. Within a service, programmatic creation of content is through [Search Service REST API](#) or an SDK such as [Azure SDK for .NET](#). While there are no dedicated commands for content migration, you can write script that calls REST API or a client library to create and load indexes on a new service.

Preview administration features are typically not available in the [az search module](#). If you want to use a preview feature, [use the Management REST API](#) and a preview API version.

Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Get started with Azure Cloud Shell](#).

 [Launch Cloud Shell](#) 

- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the `az login` command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Authenticate to Azure using Azure CLI](#).
 - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use and manage extensions with the Azure CLI](#).
 - Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Azure CLI versions are [listed on GitHub](#).

The `az search` module extends the [Azure CLI](#) with full parity to the stable versions of the [Search Management REST APIs](#).

List services in a subscription

The following commands are from [az resource](#), returning information about existing resources and services already provisioned in your subscription. If you don't know how many search services are already created, these commands return that information, saving you a trip to the Azure portal.

The first command returns all search services.

Azure CLI

```
az resource list --resource-type Microsoft.Search/searchServices --output table
```

From the list of services, return information about a specific resource.

Azure CLI

```
az resource list --name <search-service-name>
```

List all az search commands

You can view information on the subgroups and commands available in `az search` from within the CLI. Alternatively, you can review the [documentation](#).

To view the subgroups available within `az search`, run the following command.

```
Azure CLI
```

```
az search --help
```

The response should look similar to the following output.

```
Bash
```

Group

```
az search : Manage Azure Search services, admin keys and query keys.
```

```
WARNING: This command group is in preview and under development. Reference and support
```

```
levels: https://aka.ms/CLI\_refstatus
```

Subgroups:

```
admin-key : Manage Azure Search admin keys.
```

```
private-endpoint-connection : Manage Azure Search private endpoint connections.
```

```
private-link-resource : Manage Azure Search private link resources.
```

```
query-key : Manage Azure Search query keys.
```

```
service : Manage Azure Search services.
```

```
shared-private-link-resource : Manage Azure Search shared private link resources.
```

```
For more specific examples, use: az find "az search"
```

Within each subgroup, multiple commands are available. You can see the available commands for the `service` subgroup by running the following line.

```
Azure CLI
```

```
az search service --help
```

You can also see the arguments available for a particular command.

```
Azure CLI
```

```
az search service create --help
```

Get search service information

If you know the resource group containing your search service, run [az search service show](#) to return the service definition, including name, region, tier, and replica and partition counts. For this command, provide the resource group that contains the search service.

Azure CLI

```
az search service show --name <service-name> --resource-group <search-service-resource-group-name>
```

Create or delete a service

To [create a new search service](#), use the [az search service create](#) command.

Azure CLI

```
az search service create \
  --name <service-name> \
  --resource-group <search-service-resource-group-name> \
  --sku Standard \
  --partition-count 1 \
  --replica-count 1
```

Results should look similar to the following output:

```
{
  "hostingMode": "default",
  "id": "/subscriptions/<alphanumeric-subscription-ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp",
  "identity": null,
  "location": "West US",
  "name": "my-demo-searchapp",
  "networkRuleSet": {
    "bypass": "None",
    "ipRules": []
  },
  "partitionCount": 1,
  "privateEndpointConnections": [],
  "provisioningState": "succeeded",
  "publicNetworkAccess": "Enabled",
  "replicaCount": 1,
  "resourceGroup": "demo-westus",
  "sharedPrivateLinkResources": [],
  "sku": {
    "name": "standard"
  }
}
```

```
},
  "status": "running",
  "statusDetails": "",
  "tags": null,
  "type": "Microsoft.Search/searchServices"
}
```

az search service delete removes the service and its data.

Azure CLI

```
az search service delete --name <service-name> \
                        --resource-group <search-service-resource-group-name> \
```

Create a service with IP rules

Depending on your security requirements, you might want to create a search service with an [IP firewall configured](#). To do so, pass the Public IP (v4) addresses or CIDR ranges to the `ip-rules` argument as shown below. Rules should be separated by a comma (,) or semicolon (;).

Azure CLI

```
az search service create \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1 \
    --ip-rules "55.5.63.73;52.228.215.197;101.37.221.205"
```

Create a service with a system assigned managed identity

In some cases, such as when [using managed identity to connect to a data source](#), you need to turn on [system assigned managed identity](#). This is done by adding `--identity-type SystemAssigned` to the command.

Azure CLI

```
az search service create \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1 \
    --identity-type SystemAssigned
```

Create a service with a private endpoint

Private Endpoints for Azure AI Search allow a client on a virtual network to securely access data in a search index over a [Private Link](#). The private endpoint uses an IP address from the [virtual network address space](#) for your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. For more information, please refer to the documentation on [creating a private endpoint for Azure AI Search](#).

The following example shows how to create a search service with a private endpoint.

First, deploy a search service with `PublicNetworkAccess` set to `Disabled`.

Azure CLI

```
az search service create \
  --name <search-service-name> \
  --resource-group <search-service-resource-group-name> \
  --sku Standard \
  --partition-count 1 \
  --replica-count 1 \
  --public-access Disabled
```

Next, create a virtual network and the private endpoint.

Azure CLI

```
# Create the virtual network
az network vnet create \
  --resource-group <vnet-resource-group-name> \
  --location "West US" \
  --name <virtual-network-name> \
  --address-prefixes 10.1.0.0/16 \
  --subnet-name <subnet-name> \
  --subnet-prefixes 10.1.0.0/24

# Update the subnet to disable private endpoint network policies
az network vnet subnet update \
  --name <subnet-name> \
  --resource-group <vnet-resource-group-name> \
  --vnet-name <virtual-network-name> \
  --disable-private-endpoint-network-policies true

# Get the id of the search service
id=$(az search service show \
  --resource-group <search-service-resource-group-name> \
  --name <search-service-name> \
  --query [id] \
  --output tsv)
```

```
# Create the private endpoint
az network private-endpoint create \
    --name <private-endpoint-name> \
    --resource-group <private-endpoint-resource-group-name> \
    --vnet-name <virtual-network-name> \
    --subnet <subnet-name> \
    --private-connection-resource-id $id \
    --group-id searchService \
    --connection-name <private-link-connection-name>
```

Finally, create a private DNS Zone.

Azure CLI

```
## Create private DNS zone
az network private-dns zone create \
    --resource-group <private-dns-resource-group-name> \
    --name "privatelink.search.windows.net"

## Create DNS network link
az network private-dns link vnet create \
    --resource-group <private-dns-resource-group-name> \
    --zone-name "privatelink.search.windows.net" \
    --name "myLink" \
    --virtual-network <virtual-network-name> \
    --registration-enabled false

## Create DNS zone group
az network private-endpoint dns-zone-group create \
    --resource-group <private-endpoint-resource-group-name> \
    --endpoint-name <private-endpoint-name> \
    --name "myZoneGroup" \
    --private-dns-zone "privatelink.search.windows.net" \
    --zone-name "searchServiceZone"
```

For more information on creating private endpoints in Azure CLI, see this [Private Link Quickstart](#).

Manage private endpoint connections

In addition to creating a private endpoint connection, you can also `show`, `update`, and `delete` the connection.

To retrieve a private endpoint connection and to see its status, use `az search private-endpoint-connection show`.

Azure CLI

```
az search private-endpoint-connection show \
--name <pe-connection-name> \
--service-name <search-service-name> \
--resource-group <search-service-resource-group-name>
```

To update the connection, use **az search private-endpoint-connection update**. The following example sets a private endpoint connection to rejected:

Azure CLI

```
az search private-endpoint-connection update \
--name <pe-connection-name> \
--service-name <search-service-name> \
--resource-group <search-service-resource-group-name>
--status Rejected \
--description "Rejected" \
--actions-required "Please fix XYZ"
```

To delete the private endpoint connection, use **az search private-endpoint-connection delete**.

Azure CLI

```
az search private-endpoint-connection delete \
--name <pe-connection-name> \
--service-name <search-service-name> \
--resource-group <search-service-resource-group-name>
```

Create a service with confidential computing

[Confidential computing](#) is an optional compute type for data-in-use protection. When configured, your search service is deployed on confidential VMs (DCasv5 or DCesv5) instead of standard VMs. This compute type also incurs a 10% surcharge for billable tiers. For more information, see the [pricing page](#).

For daily usage, confidential computing isn't necessary. We only recommend this compute type for stringent regulatory, compliance, or security requirements. For more information, see [Confidential computing use cases](#).

The compute type is fixed for the lifetime of your search service. To permanently configure confidential computing, set the `compute-type` property to `confidential` on a new service.

Azure CLI

```
az search service create \
--name <search-service-name> \
```

```
--resource-group <search-service-resource-group-name> \
--location <search-service-region> \
--sku basic \
--compute-type confidential
```

Regenerate admin keys

To roll over admin [API keys](#), use `az search admin-key renew`. Two admin keys are created with each service for authenticated access. Keys are required on every request. Both admin keys are functionally equivalent, granting full write access to a search service with the ability to retrieve any information, or create and delete any object. Two keys exist so that you can use one while replacing the other.

You can only regenerate one at a time, specified as either the `primary` or `secondary` key. For uninterrupted service, remember to update all client code to use a secondary key while rolling over the primary key. Avoid changing the keys while operations are in flight.

As you might expect, if you regenerate keys without updating client code, requests using the old key will fail. Regenerating all new keys doesn't permanently lock you out of your service, and you can still access the service through the Azure portal. After you regenerate primary and secondary keys, you can update client code to use the new keys and operations will resume accordingly.

Values for the API keys are generated by the service. You can't provide a custom key for Azure AI Search to use. Similarly, there's no user-defined name for admin API-keys. References to the key are fixed strings, either `primary` or `secondary`.

Azure CLI

```
az search admin-key renew \
--resource-group <search-service-resource-group-name> \
--service-name <search-service-name> \
--key-kind primary
```

Results should look similar to the following output. Both keys are returned even though you only change one at a time.

```
{
  "primaryKey": <alphanumeric-guid>,
  "secondaryKey": <alphanumeric-guid>
}
```

Create or delete query keys

To create query [API keys](#) for read-only access from client apps to an Azure AI Search index, use [az search query-key create](#). Query keys are used to authenticate to a specific index for retrieving search results. Query keys don't grant read-only access to other items on the service, such as an index, data source, or indexer.

You can't provide a key for Azure AI Search to use. API keys are generated by the service.

Azure CLI

```
az search query-key create \
  --name myQueryKey \
  --resource-group <search-service-resource-group-name> \
  --service-name <search-service-name>
```

Scale replicas and partitions

To [increase or decrease replicas and partitions](#) use [az search service update](#). Increasing replicas or partitions adds to your bill, which has both fixed and variable charges. If you have a temporary need for more processing power, you can increase replicas and partitions to handle the workload. The monitoring area in the Overview portal page has tiles on query latency, queries per second, and throttling, indicating whether current capacity is adequate.

It can take a while to add or remove resourcing. Adjustments to capacity occur in the background, allowing existing workloads to continue. Extra capacity is used for incoming requests as soon as it's ready, with no extra configuration required.

Removing capacity can be disruptive. Stopping all indexing and indexer jobs prior to reducing capacity is recommended to avoid dropped requests. If that isn't feasible, you might consider reducing capacity incrementally, one replica and partition at a time, until your new target levels are reached.

Once you submit the command, there's no way to terminate it midway through. You have to wait until the command is finished before revising the counts.

Azure CLI

```
az search service update \
  --name <search-service-name> \
  --resource-group <search-service-resource-group-name> \
  --partition-count 6 \
  --replica-count 6
```

In addition to updating replica and partition counts, you can also update `ip-rules`, `public-access`, and `identity-type`.

Create a shared private link resource

Private endpoints of secured resources that are created through Azure AI Search APIs are referred to as *shared private link resources*. This is because you're "sharing" access to a resource, such as a storage account that has been integrated with the [Azure Private Link service](#).

If you're using an indexer to index data in Azure AI Search, and your data source is on a private network, you can create an outbound [private endpoint connection](#) to reach the data.

A full list of the Azure Resources for which you can create outbound private endpoints from Azure AI Search can be found [here](#) along with the related **Group ID** values.

To create the shared private link resource, use [`az search shared-private-link-resource create`](#). Keep in mind that some configuration might be required for the data source before running this command.

```
Azure CLI  
  
az search shared-private-link-resource create \  
  --name <spl-name> \  
  --service-name <search-service-name> \  
  --resource-group <search-service-resource-group-name> \  
  --group-id blob \  
  --resource-id "/subscriptions/<alphanumeric-subscription-  
ID>/resourceGroups/<resource-group-  
name>/providers/Microsoft.Storage/storageAccounts/myBlobStorage" \  
  --request-message "Please approve"
```

To retrieve the shared private link resources and view their status, use [`az search shared-private-link-resource list`](#).

```
Azure CLI  
  
az search shared-private-link-resource list \  
  --service-name <search-service-name> \  
  --resource-group <search-service-resource-group-name>
```

You need to approve the connection with the following command before it can be used. The ID of the private endpoint connection must be retrieved from the child resource. In this case, we get the connection ID from `az storage`.

Azure CLI

```
id = (az storage account show -n myBlobStorage --query  
"privateEndpointConnections[0].id")  
  
az network private-endpoint-connection approve --id $id
```

To delete the shared private link resource, use [az search shared-private-link-resource delete](#).

Azure CLI

```
az search shared-private-link-resource delete \  
--name <spl-name> \  
--service-name <search-service-name> \  
--resource-group <search-service-resource-group-name>
```

For more information on setting up shared private link resources, see [making indexer connections through a private endpoint](#).

Next steps

Build an [index](#), [query an index](#) using the Azure portal, REST APIs, or the .NET SDK.

- [Create an Azure AI Search index in the Azure portal](#)
- [Set up an indexer to load data from other services](#)
- [Query an Azure AI Search index using Search explorer in the Azure portal](#)
- [How to use Azure AI Search in .NET](#)

Manage your Azure AI Search service using REST APIs

Learn how to create and configure an Azure AI Search service using the [Management REST APIs](#). Only the Management REST APIs are guaranteed to provide early access to [preview features](#).

The Management REST API is available in stable and preview versions. Be sure to set a preview API version if you're accessing preview features.

- ✓ [Create or update a service](#)
- ✓ [Upgrade a service](#)
- ✓ [Change pricing tiers](#)
- ✓ [Configure role-based access control for data plane](#)
- ✓ [Configure confidential computing](#)
- ✓ [Enable Azure role-based access control for data plane](#)
- ✓ [Enforce a customer-managed key policy](#)
- ✓ [Disable semantic ranker](#)
- ✓ [Disable workloads that push data to external resources](#)
- ✓ [Create a query key](#)
- ✓ [Regenerate an admin key](#)
- ✓ [List private endpoint connections](#)
- ✓ [List search operations](#)
- ✓ [Delete a search service](#)

All of the Management REST APIs have examples. If a task isn't covered in this article, see the [API reference](#) instead.

💡 Tip

If you use CURL to call the Management REST API, make sure you set a content type header to application/json: `-H "Content-Type: application/json"`. Alternatively, you can use the `--JSON` flag if you want to embed the JSON.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Visual Studio Code](#) with a [REST client](#).

- Azure CLI to get an access token, as described in the following steps. You must be an owner or administrator in your Azure subscription.

Management REST API calls are authenticated through Microsoft Entra ID. You must provide an access token on the request and permissions to create and configure a resource. In addition to the Azure CLI, you can use [Azure PowerShell to create an access token](#).

1. Open a command shell for Azure CLI.
2. Sign in to your Azure subscription. If you have multiple tenants or subscriptions, make sure you select the correct one.

```
Azure CLI
```

```
az login
```

3. Get the tenant ID and subscription ID.

```
Azure CLI
```

```
az account show
```

4. Get an access token.

```
Azure CLI
```

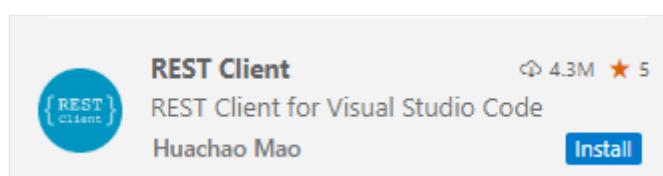
```
az account get-access-token --query accessToken --output tsv
```

You should have a tenant ID, subscription ID, and bearer token. You'll paste these values into the `.rest` or `.http` file that you create in the next step.

Set up Visual Studio Code

If you're not familiar with the REST client for Visual Studio Code, this section includes setup so that you can complete the tasks in this article.

1. Start Visual Studio Code and select the **Extensions** tile.
2. Search for the REST client and select **Install**.



3. Open or create new file named with either a `.rest` or `.http` file extension.

4. Provide variables for the values you retrieved in the previous step.

HTTP

```
@tenant-id = PUT-YOUR-TENANT-ID-HERE  
@subscription-id = PUT-YOUR-SUBSCRIPTION-ID-HERE  
@token = PUT-YOUR-TOKEN-HERE
```

5. Verify the session is operational by listing search services in your subscription.

HTTP

```
### List search services  
GET https://management.azure.com/subscriptions/{{subscription-  
id}}/providers/Microsoft.Search/searchServices?api-version=2025-05-01 HTTP/1.1  
Content-type: application/json  
Authorization: Bearer {{token}}
```

6. Select **Send request**. A response should appear in an adjacent pane. If you have existing search services, they're listed. Otherwise, the list is empty, but as long as the HTTP code is 200 OK, you're ready for the next steps.

HTTP

```
HTTP/1.1 200 OK  
Cache-Control: no-cache  
Pragma: no-cache  
Content-Length: 22068  
Content-Type: application/json; charset=utf-8  
Expires: -1  
x-ms-ratelimit-remaining-subscription-reads: 11999  
x-ms-request-id: f47d3562-a409-49d2-b9cd-6a108e07304c  
x-ms-correlation-request-id: f47d3562-a409-49d2-b9cd-6a108e07304c  
x-ms-routing-request-id: WESTUS2:20240314T012052Z:f47d3562-a409-49d2-b9cd-  
6a108e07304c  
Strict-Transport-Security: max-age=31536000; includeSubDomains  
X-Content-Type-Options: nosniff  
X-Cache: CONFIG_NOCACHE  
X-MSEdge-Ref: Ref A: 12401F1160FE4A3A8BB54D99D1FDEE4E Ref B: C06AA3150217011  
Ref C: 2024-03-14T01:20:52Z  
Date: Thu, 14 Mar 2024 01:20:52 GMT  
Connection: close  
  
{  
    "value": [ . . . ]  
}
```

Create or update a service

Creates or updates a search service under the current subscription. This example uses variables for the search service name and region, which haven't been defined yet. Either provide the names directly or add new variables to the collection.

HTTP

```
### Create a search service (provide an existing resource group)
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PUT https://management.azure.com/subscriptions/{{subscription-
id}}/resourceGroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}


{
    "location": "North Central US",
    "sku": {
        "name": "basic"
    },
    "properties": {
        "replicaCount": 1,
        "partitionCount": 1,
        "hostingMode": "default"
    }
}
```

Upgrade a service

Some Azure AI Search capabilities are only available to new services. To avoid service recreation and bring these capabilities to an existing service, you might be able to [upgrade your service](#).

HTTP

```
### Upgrade a search service
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

POST https://management.azure.com/subscriptions/{{subscription-
id}}/resourceGroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}/upgrade?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}
```

Change pricing tiers

If you need more or less capacity, you can [switch to a different pricing tier](#). Currently, you can only switch between Basic and Standard (S1, S2, and S3) tiers. Use the `sku` property to specify the new tier.

HTTP

```
### Change pricing tiers
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PATCH https://management.azure.com/subscriptions/{{subscription-
id}}/resourceGroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {token}

{
    "sku": {
        "name": "standard2"
    }
}
```

Create an S3HD service

To create an [S3HD](#) service, use a combination of `sku` and `hostingMode` properties. Set `sku` to `standard3` and "hostingMode" to `HighDensity`.

HTTP

```
### Create an S3HD service
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PUT https://management.azure.com/subscriptions/{{subscription-
id}}/resourceGroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {token}

{
    "location": "{{region}}",
    "sku": {
        "name": "standard3"
    },
    "properties": {
```

```
        "replicaCount": 1,  
        "partitionCount": 1,  
        "hostingMode": "HighDensity"  
    }  
}
```

Configure role-based access for data plane

Applies to: Search Index Data Contributor, Search Index Data Reader, Search Service Contributor

Configure your search service to recognize an **authorization** header on data requests that provide an OAuth2 access token.

To use role-based access control for data plane operations, set `authOptions` to `aadOrApiKey` and then send the request.

To use role-based access control exclusively, [turn off API key authentication](#) by following up with a second request, this time setting `disableLocalAuth` to true.

HTTP

```
### Configure role-based access  
@resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE  
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE  
  
PATCH https://management.azure.com/subscriptions/{{subscription-  
id}}/resourcegroups/{{resource-  
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-  
version=2025-05-01 HTTP/1.1  
Content-type: application/json  
Authorization: Bearer {{token}}  
  
{  
    "properties": {  
        "disableLocalAuth": false,  
        "authOptions": {  
            "aadOrApiKey": {  
                "aadAuthFailureMode": "http401WithBearerChallenge"  
            }  
        }  
    }  
}
```

Configure confidential computing

Confidential computing is an optional compute type for data-in-use protection. When configured, your search service is deployed on confidential VMs (DCasv5 or DCesv5) instead of standard VMs. This compute type also incurs a 10% surcharge for billable tiers. For more information, see the [pricing page](#).

For daily usage, confidential computing isn't necessary. We only recommend this compute type for stringent regulatory, compliance, or security requirements. For more information, see [Confidential computing use cases](#).

The compute type is fixed for the lifetime of your search service. To permanently configure confidential computing, set the `computeType` property to `confidential` on a new service.

HTTP

```
### Configure confidential computing
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE
PUT https://management.azure.com/subscriptions/{{subscription-id}}/resourcegroups/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-version=2025-05-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}
{
    "location": "{{region}}",
    "sku": {
        "name": "basic"
    },
    "properties": {
        "computeType": "confidential"
    }
}
```

Enforce a customer-managed key policy

If you're using [customer-managed encryption](#), you can enable "encryptionWithCMK" with "enforcement" set to "Enabled" if you want the search service to report its compliance status.

When you enable this policy, any REST calls that create objects containing sensitive data, such as the connection string within a data source, will fail if an encryption key isn't provided: "Error creating Data Source: "CannotCreateNonEncryptedResource: The creation of non-encrypted DataSources is not allowed when encryption policy is enforced."

HTTP

```
### Enforce a customer-managed key policy
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
```

```
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PATCH https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}


{
    "properties": {
        "encryptionWithCmk": {
            "enforcement": "Enabled"
        }
    }
}
```

Disable semantic ranker

Semantic ranker is enabled by default at the free plan that allows up to 1,000 requests per month at no charge. You can lock down the feature at the service level to prevent usage.

HTTP

```
### Disable semantic ranker
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PATCH https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}


{
    "properties": {
        "semanticSearch": "Disabled"
    }
}
```

Disable workloads that push data to external resources

Azure AI Search writes to external data sources when updating a knowledge store, saving debug session state, or caching enrichments. The following example disables these workloads at the service level.

HTTP

```
### Disable external access
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

PATCH https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}


{
    "properties": {
        "publicNetworkAccess": "Disabled"
    }
}
```

Delete a search service

HTTP

```
### Delete a search service
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

DELETE https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}?api-
version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}
```

List admin API keys

HTTP

```
### List admin keys
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

POST https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service}}/listAdminKeys?
api-version=2025-05-01 HTTP/1.1
    Content-type: application/json
    Authorization: Bearer {{token}}
```

Regenerate admin API keys

You can only regenerate one admin API key at a time.

HTTP

```
### Regnerate admin keys
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

POST https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-
service}}/regenerateAdminKey/primary?api-version=2025-05-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}
```

Create query API keys

HTTP

```
### Create a query key
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE
@query-key = PUT-YOUR-QUERY-KEY-NAME-HERE

POST https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-
service}}/createQueryKey/{query-key}?api-version=2025-05-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}
```

List private endpoint connections

HTTP

```
### List private endpoint connections
@Resource-group = PUT-YOUR-RESOURCE-GROUP-NAME-HERE
@search-service = PUT-YOUR-SEARCH-SERVICE-NAME-HERE

GET https://management.azure.com/subscriptions/{{subscription-
id}}/resourcegroups/{{resource-
group}}/providers/Microsoft.Search/searchServices/{{search-
service}}/privateEndpointConnections?api-version=2025-05-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}
```

List search operations

HTTP

```
### List search operations
GET https://management.azure.com/subscriptions/{{subscription-id}}/resourcegroups?
api-version=2021-04-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}
```

Next steps

After a search service is configured, your next steps include [creating an index](#) or [querying an index](#) using the Azure portal, REST APIs, or an Azure SDK.

- [Create an Azure AI Search index in the Azure portal](#)
- [Set up an indexer to load data from other services](#)
- [Query an Azure AI Search index using Search explorer in the Azure portal](#)
- [How to use Azure AI Search in .NET](#)

Last updated on 12/04/2025

Move your Azure AI Search service to another Azure region

Occasionally, customers ask about moving a search service to another region. Currently, there's no built-in mechanism or tooling to help with that task, but this article can help you understand the manual steps for recreating indexes and other objects on a new search service in a different region.

(!) Note

In the Azure portal, all services have an **Export template** command. In the case of Azure AI Search, this command produces a basic definition of a service (name, location, tier, replica, and partition count), but does not recognize the content of your service, nor does it carry over keys, roles, or logs. Although the command exists, we don't recommend using it for moving a search service.

Prerequisites

- Ensure that the services and features that your account uses are supported in the target region.
- For preview features, ensure that your subscription is approved for the target region.

Prepare and move

1. Identify dependencies and related services to understand the full impact of relocating a service, in case you need to move more than just Azure AI Search.

Azure Storage is used for logging, creating a knowledge store, and is a commonly used external data source for AI enrichment and indexing. Foundry Tools are used to power built-in skills during AI enrichment. Both Foundry Tools and your search service are required to be in the same region if you're using AI enrichment.

2. Create an inventory of all objects on the service so that you know what to move: indexes, synonym maps, indexers, data sources, skillsets. If you enabled logging, create and archive any reports you might need for a historical record.
3. Check pricing and availability in the new region to ensure availability of Azure AI Search plus any related services in the new region. Most features are available in all regions, but some preview features have restricted availability.

4. Create a service in the new region and republish from source code any existing indexes, synonym maps, indexers, data sources, and skillsets. Remember that service names must be unique so you can't reuse the existing name. Check each skillset to see if connections to Foundry Tools are still valid in terms of the same-region requirement. Also, if knowledge stores are created, check the connection strings for Azure Storage if you're using a different service.
5. Reload indexes and knowledge stores, if applicable. You'll either use application code to push JSON data into an index, or rerun indexers to pull documents in from external sources.
6. Enable logging, and if you're using them, re-create security roles.
7. Update client applications and test suites to use the new service name and API keys, and test all applications.

Discard or clean up

Delete the old service once the new service is fully tested and operational. Deleting the service automatically deletes all content associated with the service.

Next steps

The following links can help you locate more information when completing the steps outlined above.

- [Azure AI Search pricing and regions ↗](#)
- [Choose a tier](#)
- [Create a search service](#)
- [Load search documents](#)
- [Enable logging](#)

Create an index in Azure AI Search

08/27/2025

In this article, learn the steps for defining a schema for a [search index](#) and pushing it to a search service. Creating an index establishes the physical data structures on your search service. Once the index exists, [load the index](#) as a separate task.

Prerequisites

- Write permissions as a [Search Service Contributor](#) or an [admin API key](#) for key-based authentication.
- An understanding of the data you want to index. A search index is based on external content that you want to make searchable. Searchable content is stored as fields in an index. You should have a clear idea of which source fields you want to make searchable, retrievable, filterable, facetable, and sortable on Azure AI Search. See the [schema checklist](#) for guidance.
- You must also have a unique field in source data that can be used as the [document key \(or ID\)](#) in the index.
- A stable index location. Moving an existing index to a different search service isn't supported out-of-the-box. Revisit application requirements and make sure that your existing search service (capacity and region), are sufficient for your needs. If you're taking a dependency on Azure AI services or Azure OpenAI, [choose a region](#) that provides all of the necessary resources.
- Finally, all service tiers have [index limits](#) on the number of objects that you can create. For example, if you're experimenting on the Free tier, you can only have three indexes at any given time. Within the index itself, there are [limits on vectors](#) and [index limits](#) on the number of simple and complex fields.

Document keys

Search index creation has two requirements: an index must have a unique name on the search service, and it must have a document key. The boolean `key` attribute on a field can be set to true to indicate which field provides the document key.

A document key is the unique identifier of a search document, and a search document is a collection of fields that completely describes something. For example, if you're indexing a [movies data set](#), a search document contains the title, genre, and duration of a single movie.

Movie names are unique in this dataset, so you might use the movie name as the document key.

In Azure AI Search, a document key is a string, and it must originate from unique values in the data source that's providing the content to be indexed. As a general rule, a search service doesn't generate key values, but in some scenarios (such as the [Azure table indexer](#)) it synthesizes existing values to create a unique key for the documents being indexed. Another scenario is one-to-many indexing for chunked or partitioned data, in which case document keys are generated for each chunk.

During incremental indexing, where new and updated content is indexed, incoming documents with new keys are added, while incoming documents with existing keys are either merged or overwritten, depending on whether index fields are null or populated.

Important points about document keys include:

- The maximum length of values in a key field is 1,024 characters.
- Exactly one top-level field in each index must be chosen as the key field and it must be of type `Edm.String`.
- The default of the `key` attribute is false for simple fields and null for complex fields.

Key fields can be used to look up documents directly and update or delete specific documents. The values of key fields are handled in a case-sensitive manner when looking up or indexing documents. See [GET Document \(REST\)](#) and [Index Documents \(REST\)](#) for details.

Schema checklist

Use this checklist to assist the design decisions for your search index.

1. Review [naming conventions](#) so that index and field names conform to the naming rules.
2. Review [supported data types](#). The data type affects how the field is used. For example, numeric content is filterable but not full text searchable. The most common data type is `Edm.String` for searchable text, which is tokenized and queried using the full text search engine. The most common data type for a vector field is `Edm.Single` but you can use other types as well.
3. Provide a description of the index, 4,000 character maximum. This human-readable text is invaluable when a system must access several indexes and make a decision based on the description. Consider a Model Context Protocol (MCP) server that must pick the correct index at run time. The decision can be based on the description rather than on index name alone.

4. Identify a [document key](#). A document key is an index requirement. It's a single string field populated from a source data field that contains unique values. For example, if you're indexing from Blob Storage, the metadata storage path is often used as the document key because it uniquely identifies each blob in the container.

5. Identify the fields in your data source that contribute searchable content in the index.

Searchable nonvector content includes short or long strings that are queried using the full text search engine. If the content is verbose (small phrases or bigger chunks), experiment with different analyzers to see how the text is tokenized.

Searchable vector content can be images or text (in any language) that exists as a mathematical representation. You can use narrow data types or vector compression to make vector fields smaller.

[Attributes set on fields](#), such as `retrievable` or `filterable`, determine both search behaviors and the physical representation of your index on the search service. Determining how fields should be attributed is an iterative process for many developers. To speed up iterations, start with sample data so that you can drop and rebuild easily.

6. Identify which source fields can be used as filters. Numeric content and short text fields, particularly those with repeating values, are good choices. When working with filters, remember:

- Filters can be used in vector and nonvector queries, but the filter itself is applied to human-readable (nonvector) fields in your index.
- Filterable fields can optionally be used in faceted navigation.
- Filterable fields are returned in arbitrary order and don't undergo relevance scoring, so consider making them sortable as well.

7. For vector fields, specify a vector search configuration and the algorithms used for creating navigation paths and filling the embedding space. For more information, see [Add vector fields](#).

Vector fields have extra properties that nonvector fields don't have, such as which algorithms to use and vector compression.

Vector fields omit attributes that aren't useful on vector data, such as sorting, filtering, and facetting.

8. For nonvector fields, determine whether to use the default analyzer (`"analyzer": null`) or a different analyzer. [Analyzers](#) are used to tokenize text fields during indexing and query execution.

For multi-lingual strings, consider a [language analyzer](#).

For hyphenated strings or special characters, consider [specialized analyzers](#). One example is [keyword](#) that treats the entire contents of a field as a single token. This behavior is useful for data like zip codes, IDs, and some product names. For more information, see [Partial term search and patterns with special characters](#).

Note

Full text search is conducted over terms that are tokenized during indexing. If your queries fail to return the results you expect, [test for tokenization](#) to verify the string you're searching for actually exists. You can try different analyzers on strings to see how tokens are produced for various analyzers.

Configure field definitions

The fields collection defines the structure of a search document. All fields have a name, data type, and attributes.

Setting a field as searchable, filterable, sortable, or facetable has an effect on index size and query performance. Don't set those attributes on fields that aren't meant to be referenced in query expressions.

If a field isn't set to be searchable, filterable, sortable, or facetable, the field can't be referenced in any query expression. This is desirable for fields that aren't used in queries, but are needed in search results.

The REST APIs have default attribution based on [data types](#), which is also used by the [import wizards](#) in the Azure portal. The Azure SDKs don't have defaults, but they have field subclasses that incorporate properties and behaviors, such as [SearchableField](#) for strings and [SimpleField](#) for primitives.

Default field attributions for the REST APIs are summarized in the following table.

 Expand table

Data type	Searchable	Retrievable	Filterable	Facetable	Sortable	Stored
Edm.String						
Collection(Edm.String)						
Edm.Boolean						

Data type	Searchable	Retrievable	Filterable	Facetable	Sortable	Stored
<code>Edm.Int32</code> , <code>Edm.Int64</code> , <code>Edm.Double</code>	✗	✓	✓	✓	✓	✓
<code>Edm.DateTimeOffset</code>	✗	✓	✓	✓	✓	✓
<code>Edm.GeographyPoint</code>	✓	✓	✓	✗	✓	✓
<code>Edm.ComplexType</code>	✓	✓	✓	✓	✓	✓
<code>Collection(Edm.Single)</code> and all other vector field types	✓	✓ or ✗	✗	✗	✗	✓

String fields can also be optionally associated with [analyzers](#) and [synonym maps](#). Fields of type `Edm.String` that are filterable, sortable, or facetable can be at most 32 kilobytes in length. This is because values of such fields are treated as a single search term, and the maximum length of a term in Azure AI Search is 32 kilobytes. If you need to store more text than this in a single string field, you should explicitly set filterable, sortable, and facetable to `false` in your index definition.

Vector fields must be associated with [dimensions](#) and [vector profiles](#). Retrievable is true by default if you add the vector field using the [Import data \(new\) wizard](#) in the Azure portal. If you use the REST API, it's false.

Field attributes are described in the following table.

[] Expand table

Attribute	Description
<code>name</code>	Required. Sets the name of the field, which must be unique within the fields collection of the index or parent field.
<code>type</code>	Required. Sets the data type for the field. Fields can be simple or complex. Simple fields are of primitive types, like <code>Edm.String</code> for text or <code>Edm.Int32</code> for integers. Complex fields can have sub-fields that are themselves either simple or complex. This allows you to model objects and arrays of objects, which in turn enables you to upload most JSON object structures to your index. See Supported data types for the complete list of supported types.
<code>key</code>	Required. Set this attribute to true to designate that a field's values uniquely identify documents in the index. See Document keys in this article for details.
<code>retrievable</code>	Indicates whether the field can be returned in a search result. Set this attribute to <code>false</code> if you want to use a field as a filter, sorting, or scoring mechanism but don't want the field to be visible to the end user. This attribute must be <code>true</code> for key fields, and it must be <code>null</code> for complex fields. This attribute can be changed on existing fields. Setting

Attribute	Description
	retrievable to <code>true</code> doesn't cause any increase in index storage requirements. Default is <code>true</code> for simple fields and <code>null</code> for complex fields.
searchable	<p>Indicates whether the field is full-text searchable and can be referenced in search queries. This means it undergoes lexical analysis such as word-breaking during indexing. If you set a searchable field to a value like "Sunny day", internally it's normalized into the individual tokens "sunny" and "day". This enables full-text searches for these terms. Fields of type <code>Edm.String</code> or <code>Collection(Edm.String)</code> are searchable by default. This attribute must be <code>false</code> for simple fields of other nonstring data types, and it must be <code>null</code> for complex fields.</p>
	<p>A searchable field consumes extra space in your index since Azure AI Search processes the contents of those fields and organize them in auxiliary data structures for performant searching. If you want to save space in your index and you don't need a field to be included in searches, set searchable to <code>false</code>. See How full-text search works in Azure AI Search for details.</p>
filterable	<p>Indicates whether to enable the field to be referenced in <code>\$filter</code> queries. Filterable differs from searchable in how strings are handled. Fields of type <code>Edm.String</code> or <code>Collection(Edm.String)</code> that are filterable don't undergo lexical analysis, so comparisons are for exact matches only. For example, if you set such a field <code>f</code> to "Sunny day", <code>\$filter=f eq 'sunny'</code> finds no matches, but <code>\$filter=f eq 'Sunny day'</code> will. This attribute must be <code>null</code> for complex fields. Default is <code>true</code> for simple fields and <code>null</code> for complex fields. To reduce index size, set this attribute to <code>false</code> on fields that you won't be filtering on.</p>
sortable	<p>Indicates whether to enable the field to be referenced in <code>\$orderby</code> expressions. By default Azure AI Search sorts results by score, but in many experiences users want to sort by fields in the documents. A simple field can be sortable only if it's single-valued (it has a single value in the scope of the parent document).</p> <p>Simple collection fields can't be sortable, since they're multi-valued. Simple subfields of complex collections are also multi-valued, and therefore can't be sortable. This is true whether it's an immediate parent field, or an ancestor field, that's the complex collection. Complex fields can't be sortable and the sortable attribute must be <code>null</code> for such fields. The default for sortable is <code>true</code> for single-valued simple fields, <code>false</code> for multi-valued simple fields, and <code>null</code> for complex fields.</p>
facetable	<p>Indicates whether to enable the field to be referenced in facet queries. Typically used in a presentation of search results that includes hit count by category (for example, search for digital cameras and see hits by brand, by megapixels, by price, and so on). This attribute must be <code>null</code> for complex fields. Fields of type <code>Edm.GeographyPoint</code> or <code>Collection(Edm.GeographyPoint)</code> can't be facetable. Default is <code>true</code> for all other simple fields. To reduce index size, set this attribute to <code>false</code> on fields that you won't be faceting on.</p>
analyzer	<p>Sets the lexical analyzer for tokenizing strings during indexing and query operations. Valid values for this property include language analyzers, built-in analyzers, and custom</p>

Attribute	Description
	<code>analyzers</code> . The default is <code>standard.lucene</code> . This attribute can only be used with searchable string fields, and it can't be set together with either <code>searchAnalyzer</code> or <code>indexAnalyzer</code> . Once the analyzer is chosen and the field is created in the index, it can't be changed for the field. Must be <code>null</code> for complex fields .
<code>searchAnalyzer</code>	Set this property together with <code>indexAnalyzer</code> to specify different lexical analyzers for indexing and queries. If you use this property, set <code>analyzer</code> to <code>null</code> and make sure <code>indexAnalyzer</code> is set to an allowed value. Valid values for this property include built-in analyzers and custom analyzers. This attribute can be used only with searchable fields. The search analyzer can be updated on an existing field since it's only used at query-time. Must be <code>null</code> for complex fields].
<code>indexAnalyzer</code>	Set this property together with <code>searchAnalyzer</code> to specify different lexical analyzers for indexing and queries. If you use this property, set <code>analyzer</code> to <code>null</code> and make sure <code>searchAnalyzer</code> is set to an allowed value. Valid values for this property include built-in analyzers and custom analyzers. This attribute can be used only with searchable fields. Once the index analyzer is chosen, it can't be changed for the field. Must be <code>null</code> for complex fields.
<code>synonymMaps</code>	A list of the names of synonym maps to associate with this field. This attribute can be used only with searchable fields. Currently only one synonym map per field is supported. Assigning a synonym map to a field ensures that query terms targeting that field are expanded at query-time using the rules in the synonym map. This attribute can be changed on existing fields. Must be <code>null</code> or an empty collection for complex fields.
<code>fields</code>	A list of subfields if this is a field of type <code>Edm.ComplexType</code> or <code>Collection(Edm.ComplexType)</code> . Must be <code>null</code> or empty for simple fields. See How to model complex data types in Azure AI Search for more information on how and when to use subfields.

Create an index

When you're ready to create the index, use a search client that can send the request. You can use the Azure portal or REST APIs for early development and proof-of-concept testing, otherwise it's common to use the Azure SDKs.

During development, plan on frequent rebuilds. Because physical structures are created in the service, [dropping and re-creating indexes](#) is necessary for many modifications. You might consider working with a subset of your data to make rebuilds go faster.

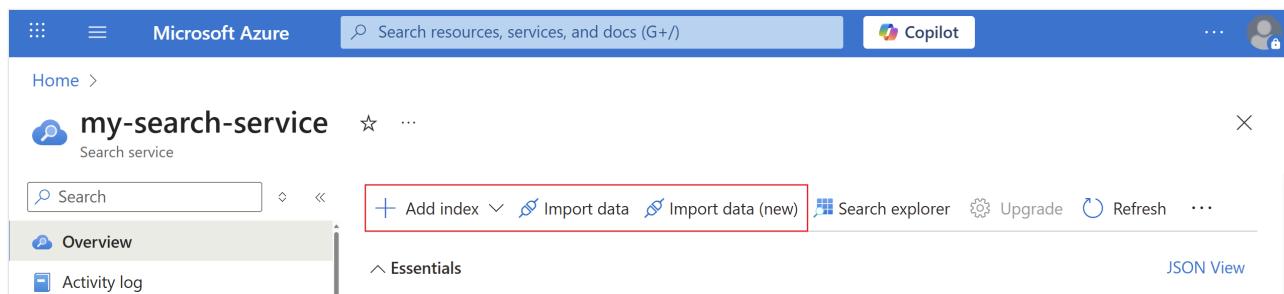
Azure portal

Index design through the Azure portal enforces requirements and schema rules for specific data types, such as disallowing full text search capabilities on numeric fields.

1. Sign in to the [Azure portal](#).
2. Check for space. Search services are subject to [maximum number of indexes](#), varying by service tier. Make sure you have room for a second index.
3. In the search service **Overview** page, choose either option for creating a search index:
 - **Add index**, an embedded editor for specifying an index schema
 - [Import wizards](#)

The wizard is an end-to-end workflow that creates an indexer, a data source, and a finished index. It also loads the data. If this is more than what you want, use **Add index** instead.

The following screenshot highlights where the **Add index**, **Import data**, and **Import data (new)** wizards appear on the command bar.



After an index is created, you can find it again on the **Indexes** page from the left pane.

Tip

After creating an index in the Azure portal, you can copy the JSON representation and add it to your application code.

Set `corsOptions` for cross-origin queries

Index schemas include a section for setting `corsOptions`. By default, client-side JavaScript can't call any APIs because browsers prevent all cross-origin requests. To allow cross-origin queries through to your index, enable CORS (Cross-Origin Resource Sharing) by setting the `corsOptions` attribute. For security reasons, only [query APIs](#) support CORS.

JSON

```
"corsOptions": {  
  "allowedOrigins": [
```

```
  ],
  "maxAgeInSeconds": 300
```

The following properties can be set for CORS:

- **allowedOrigins** (required): This is a list of origins that are allowed access to your index. JavaScript code served from these origins is allowed to query your index (assuming the caller provides a valid key or has permissions). Each origin is typically of the form `protocol://<fully-qualified-domain-name>:<port>` although `<port>` is often omitted. For more information, see [Cross-origin resource sharing \(Wikipedia\)](#).

If you want to allow access to all origins, include `*` as a single item in the **allowedOrigins** array. *This isn't a recommended practice for production search services* but it's often useful for development and debugging.

- **maxAgeInSeconds** (optional): Browsers use this value to determine the duration (in seconds) to cache CORS preflight responses. This must be a non-negative integer. A longer cache period delivers better performance, but it extends the amount of time a CORS policy needs to take effect. If this value isn't set, a default duration of five minutes is used.

Allowed updates on existing indexes

[Create Index](#) creates the physical data structures (files and inverted indexes) on your search service. Once the index is created, your ability to effect changes using [Create or Update Index](#) is contingent upon whether your modifications invalidate those physical structures. Most field attributes can't be changed once the field is created in your index.

To minimize churn in application code, you can [create an index alias](#) that serves as a stable reference to the search index. Instead of updating your code with index names, you can update an index alias to point to newer index versions.

To minimize churn in the design process, the following table describes which elements are fixed and flexible in the schema. Changing a fixed element requires an index rebuild, whereas flexible elements can be changed at any time without impacting the physical implementation. For more information, see [Update or rebuild an index](#).

[] [Expand table](#)

Element	Can be updated?
Name	No

Element	Can be updated?
Key	No
Field names and types	No
Field attributes (searchable, filterable, facetable, sortable)	No
Field attribute (retrievable)	Yes
Stored (applies to vectors)	No
Analyzer	You can add and modify custom analyzers in the index. Regarding analyzer assignments on string fields, you can only modify <code>searchAnalyzer</code> . All other assignments and modifications require a rebuild.
Scoring profiles	Yes, you can create and edit scoring profiles with no rebuild.
Suggesters	No
cross-origin resource sharing (CORS)	Yes
Encryption	Yes, you can update all parts of an <i>existing</i> encryption definition.
Synonym maps	Yes, you can create and edit synonym maps with no rebuild.
Semantic configuration	Yes, you can create and edit semantic configurations with no rebuild.

Next steps

Use the following links to learn about specialized features that can be added to an index:

- [Add vector fields and vector profiles](#)
- [Add scoring profiles](#)
- [Add semantic ranking](#)
- [Add suggesters](#)
- [Add synonym maps](#)
- [Add analyzers](#)
- [Add encryption](#)

Use these links for loading or updating an index:

- [Data import overview](#)
- [Load documents](#)
- [Update or rebuild an index](#)

Load data into a search index in Azure AI Search

10/03/2025

This article explains how to import documents into a predefined search index. In Azure AI Search, a [search index is created first](#) with [data import](#) following as a second step. The exception is [Import wizards](#) in the Azure portal and [indexer pipelines](#), which create and load an index in one workflow.

How data import works

A search service accepts JSON documents that conform to the index schema. A search service can import and index plain text content and vector content in JSON documents.

- Plain text content is retrieved from fields in the external data source, from metadata properties, or from enriched content that's generated by a [skillset](#). Skills can extract or infer textual descriptions from images and unstructured content.
- Vector content is retrieved from a data source that provides it, or it's created by a skillset that implements [integrated vectorization](#) in an Azure AI Search indexer workload.

You can prepare these documents yourself, but if content resides in a [supported data source](#), running an [indexer](#) or using an Import wizard can automate document retrieval, JSON serialization, and indexing.

Once data is indexed, the physical data structures of the index are locked in. For guidance on what can and can't be changed, see [Update and rebuild an index](#).

Indexing isn't a background process. A search service balances indexing and query workloads, but if [query latency is too high](#), you can either [add capacity](#) or identify periods of low query activity for loading an index.

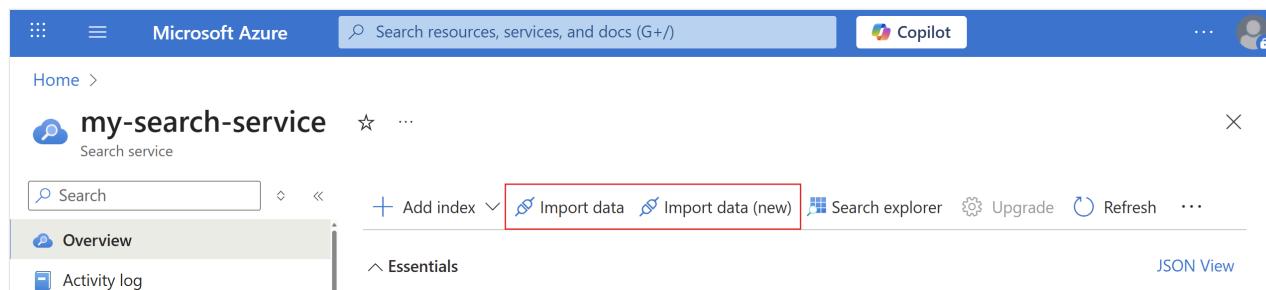
For more information, see [Data import strategies](#).

Use the Azure portal

In the Azure portal, use an [import wizard](#) to create and load indexes in a seamless workflow. If you want to load an existing index, choose an alternative approach.

1. Sign in to the [Azure portal](#) with your Azure account and [find your search service](#).

2. On the **Overview** page, select **Import data** or **Import data (new)** on the command bar to create and populate a search index.



The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header bar with the Microsoft Azure logo, a search bar containing 'Search resources, services, and docs (G+)', and a 'Copilot' button. Below the header, the URL 'Home > my-search-service' is visible. The main content area shows a 'my-search-service' search service card with a 'Search service' icon. Below the card, there are two buttons: '+ Add index' and 'Import data'. The 'Import data' button is highlighted with a red box. Other buttons in the row include 'Import data (new)', 'Search explorer', 'Upgrade', 'Refresh', and three dots. A sidebar on the left has 'Overview' selected, and a bottom right corner has a 'JSON View' link.

You can follow these links to review the workflow: [Quickstart: Create an Azure AI Search index](#) and [Quickstart: Integrated vectorization](#).

3. After the wizard is finished, use [Search Explorer](#) to check for results.

💡 Tip

The import wizards create and run indexers. If indexers are already defined, you can [reset and run an indexer](#) from the Azure portal, which is useful if you're adding fields incrementally. Reset forces the indexer to start over, picking up all fields from all source documents.

Use the REST APIs

[Documents - Index](#) is the REST API for importing data into a search index.

The body of the request contains one or more documents to be indexed. Documents are uniquely identified through a case-sensitive key. Each document is associated with an action: "upload", "delete", "merge", or "mergeOrUpload". Upload requests must include the document data as a set of key/value pairs.

REST APIs are useful for initial proof-of-concept testing, where you can test indexing workflows without having to write much code. The `@search.action` parameter determines whether documents are added in full, or partially in terms of new or replacement values for specific fields.

[Quickstart: Full-text search using REST](#) explains the steps. The following example is a modified version of the example. The value is trimmed for brevity and the first HotelId value is altered to avoid overwriting an existing document.

1. Formulate a POST call specifying the index name, the "docs/index" endpoint, and a request body that includes the `@search.action` parameter.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/index?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]
{
  "value": [
    {
      "@search.action": "upload",
      "HotelId": "1111",
      "HotelName": "Stay-Kay City Hotel",
      "Description": "The hotel is ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Time's Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.",
      "Category": "Boutique",
      "Tags": [ "pool", "air conditioning", "concierge" ]
    },
    {
      "@search.action": "mergeOrUpload",
      "HotelId": "2",
      "HotelName": "Old Century Hotel",
      "Description": "This is description is replacing the original one for this hotel. New and changed values overwrite the previous ones. In a comma-delimited list like Tags, be sure to provide the full list because there is no merging of values within the field itself.",
      "Category": "Boutique",
      "Tags": [ "pool", "free wifi", "concierge", "my first new tag", "my second new tag" ]
    }
  ]
}
```

2. Set the `@search.action` parameter to `upload` to create or overwrite a document. Set it to `merge` or `uploadOrMerge` if you're targeting updates to specific fields within the document. The previous example shows both actions.

 [Expand table](#)

Action	Effect
upload	Similar to an "upsert" where the document is inserted if it's new, and updated or replaced if it exists. If the document is missing values that the index requires, the document field's value is set to null.
merge	Updates a document that already exists, and fails a document that can't be found. Merge replaces existing values. For this reason, be sure to check for collection fields that contain multiple values, such as fields of type <code>Collection(Edm.String)</code> . For example, if a <code>tags</code> field starts with a value of

Action	Effect
	<code>["budget"]</code> and you execute a merge with <code>["economy", "pool"]</code> , the final value of the <code>tags</code> field is <code>["economy", "pool"]</code> . It isn't <code>["budget", "economy", "pool"]</code> .
mergeOrUpload	Behaves like merge if the document exists, and upload if the document is new. This is the most common action for incremental updates.
delete	Delete removes the specified document from the index. Any field you specify in a delete operation, other than the key field, is ignored. If you want to remove an individual field from a document, use merge instead and set the field explicitly to null.

There are no ordering guarantees for which action in the request body is executed first. It's not recommended to have multiple "merge" actions associated with the same document in a single request body. If there are multiple "merge" actions required for the same document, perform the merging client-side before updating the document in the search index.

In primitive collections, if the document contains a Tags field of type `Collection(Edm.String)` with a value of `["budget"]`, and you execute a merge with a value of `["economy", "pool"]` for Tag, the final value of the Tags field will be `["economy", "pool"]`. It isn't `["budget", "economy", "pool"]`.

In complex collections, if the document contains a complex collection field named Rooms with a value of `[{ "Type": "Budget Room", "BaseRate": 75.0 }]`, and you execute a merge with a value of `[{ "Type": "Standard Room" }, { "Type": "Budget Room", "BaseRate": 60.5 }]`, the final value of the Rooms field will be `[{ "Type": "Standard Room" }, { "Type": "Budget Room", "BaseRate": 60.5 }]`. It won't be either of the following:

- `[{ "Type": "Budget Room", "BaseRate": 75.0 }, { "Type": "Standard Room" }, { "Type": "Budget Room", "BaseRate": 60.5 }]` (append elements)
- `[{ "Type": "Standard Room", "BaseRate": 75.0 }, { "Type": "Budget Room", "BaseRate": 60.5 }]` (merge elements in order, then append any extras)

ⓘ Note

When you upload `DateTimeOffset` values with time zone information to your index, Azure AI Search normalizes these values to UTC. For example, `2025-01-13T14:03:00-08:00` will be stored as `2025-01-13T22:03:00Z`. If you need to store time zone information, add an extra column to your index.

3. Send the request.

The following table explains the various per-document [status codes](#) that can be returned in the response. Some status codes indicate problems with the request itself, while others indicate temporary error conditions. The latter you should retry after a delay.

 [Expand table](#)

Status code	Meaning	Retryable	Notes
200	Document was successfully modified or deleted.	n/a	Delete operations are idempotent . That is, even if a document key doesn't exist in the index, attempting a delete operation with that key results in a 200 status code.
201	Document was successfully created.	n/a	
400	There was an error in the document that prevented it from being indexed.	No	The error message in the response indicates what is wrong with the document.
404	The document couldn't be merged because the given key doesn't exist in the index.	No	This error doesn't occur for uploads since they create new documents, and it doesn't occur for deletes because they're idempotent .
409	A version conflict was detected when attempting to index a document.	Yes	This can happen when you're trying to index the same document more than once concurrently.
422	The index is temporarily unavailable because it was updated with the 'allowIndexDowntime' flag set to 'true'.	Yes	
429	Indicates that you have exceeded your quota on the number of documents per index.	No	You must either create a new index or upgrade for higher capacity limits.
503	Your search service is temporarily unavailable, possibly due to heavy load.	Yes	Your code should wait before retrying in this case or you risk prolonging the service unavailability.

 **Note**

If your client code frequently encounters a 207 response, one possible reason is that the system is under load. You can confirm this by checking the `statusCode` property for 503. If this is the case, we recommend throttling indexing requests. Otherwise, if

indexing traffic doesn't subside, the system could start rejecting all requests with 503 errors.

4. Look up the documents you just added as a validation step:

HTTP

```
GET https://[service name].search.windows.net/indexes/hotel-sample-index/docs/1111?api-version=2025-09-01
```

When the document key or ID is new, **null** becomes the value for any field that's unspecified in the document. For actions on an existing document, updated values replace the previous values. Any fields that weren't specified in a "merge" or "mergeUpload" are left intact in the search index.

Use the Azure SDKs

Programmability is provided in the following Azure SDKs.

.NET

The Azure SDK for .NET provides the following APIs for simple and bulk document uploads into an index:

- [IndexDocumentsAsync](#)
- [SearchIndexingBufferedSender](#)

There are several samples that illustrate indexing in context of simple and large-scale indexing:

- "[Load an index](#)" explains basic steps.
- [Azure.Search.Documents Samples - Indexing Documents](#) ↗ from the Azure SDK team adds [SearchIndexingBufferedSender](#).
- [Tutorial: Index any data](#) couples batch indexing with testing strategies for determining an optimum size.
- Be sure to check the [azure-search-vector-samples](#) ↗ repo for code examples showing how to index vector fields.

See also

- [Search indexes overview](#)
- [Data import overview](#)
- [Import data wizard overview](#)
- [Indexers overview](#)

Update or rebuild an index in Azure AI Search

09/28/2025

This article explains how to update an existing index in Azure AI Search with schema changes or content changes through incremental indexing. It explains the circumstances under which rebuilds are required, and provides recommendations for mitigating the effects of rebuilds on ongoing query requests.

During active development, it's common to drop and rebuild indexes when you're iterating over index design. Most developers work with a small representative sample of their data so that reindexing goes faster.

For schema changes on applications already in production, we recommend creating and testing a new index that runs side by side an existing index. Use an [index alias](#) to swap in the new index so that you can avoid changes your application code.

Update content

Incremental indexing and synchronizing an index against changes in source data is fundamental to most search applications. This section explains the workflow for adding, removing, or overwriting the content of a search index through the REST API, but the Azure SDKs provide equivalent functionality.

The body of the request contains one or more documents to be indexed. Within the request, each document in the index is:

- Identified by a unique case-sensitive key.
- Associated with an action: "upload", "delete", "merge", or "mergeOrUpload".
- Populated with a set of name/value pairs for each field that you're adding or updating.

JSON

```
{  
  "value": [  
    {  
      "@search.action": "upload (default) | merge | mergeOrUpload | delete",  
      "key_field_name": "unique_key_of_document", (key/value pair for key field  
from index schema)  
      "field_name": field_value (name/value pairs matching index schema)  
      ...  
    },  
    ...  
  ]  
}
```

```
]  
}
```

- First, use the APIs for loading documents, such as [Documents - Index \(REST\)](#) or an equivalent API in the Azure SDKs. For more information about indexing techniques, see [Load documents](#).
- For a large update, batching (up to 1,000 documents per batch, or about 16 MB per batch, whichever limit comes first) is recommended and significantly improves indexing performance.
- Set the `@search.action` parameter on the API to determine the effect on existing documents.

 [Expand table](#)

Action	Effect
delete	Removes the entire document from the index. If you want to remove an individual field, use merge instead, setting the field in question to null. Deleted documents and fields don't immediately free up space in the index. Every few minutes, a background process performs the physical deletion. Whether you use the Azure portal or an API to return index statistics, you can expect a small delay before the deletion is reflected in the Azure portal and through APIs.
merge	Updates a document that already exists, and fails a document that can't be found. Merge replaces existing values. For this reason, be sure to check for collection fields that contain multiple values, such as fields of type <code>Collection(Edm.String)</code> . For example, if a <code>tags</code> field starts with a value of <code>["budget"]</code> and you execute a merge with <code>["economy", "pool"]</code> , the final value of the <code>tags</code> field is <code>["economy", "pool"]</code> . It won't be <code>["budget", "economy", "pool"]</code> . The same behavior applies to complex collections. If the document contains a complex collection field named <code>Rooms</code> with a value of <code>[{ "Type": "Budget Room", "BaseRate": 75.0 }]</code> , and you execute a merge with a value of <code>[{ "Type": "Standard Room" }, { "Type": "Budget Room", "BaseRate": 60.5 }]</code> , the final value of the <code>Rooms</code> field will be <code>[{ "Type": "Standard Room" }, { "Type": "Budget Room", "BaseRate": 60.5 }]</code> . It won't append or merge new and existing values.
mergeOrUpload	Behaves like merge if the document exists, and upload if the document is new. This is the most common action for incremental updates.
upload	Similar to an "upsert" where the document is inserted if it's new, and updated or replaced if it exists. If the document is missing values that the index requires, the document field's value is set to null.

Queries continue to run during indexing, but if you're updating or removing existing fields, you can expect mixed results and a higher incidence of throttling.

! Note

There are no ordering guarantees for which action in the request body is executed first. It's not recommended to have multiple "merge" actions associated with the same document in a single request body. If there are multiple "merge" actions required for the same document, perform the merging client-side before updating the document in the search index.

Responses

Status code 200 is returned for a successful response, meaning that all items have been stored durably and will start to be indexed. Indexing runs in the background and makes new documents available (that is, queryable and searchable) a few seconds after the indexing operation completed. The specific delay depends on the load on the service.

Successful indexing is indicated by the `status` property being set to true for all items, as well as the `statusCode` property being set to either 201 (for newly uploaded documents) or 200 (for merged or deleted documents):

JSON

```
{
  "value": [
    {
      "key": "unique_key_of_new_document",
      "status": true,
      "errorMessage": null,
      "statusCode": 201
    },
    {
      "key": "unique_key_of_merged_document",
      "status": true,
      "errorMessage": null,
      "statusCode": 200
    },
    {
      "key": "unique_key_of_deleted_document",
      "status": true,
      "errorMessage": null,
      "statusCode": 200
    }
  ]
}
```

Status code 207 is returned when at least one item wasn't successfully indexed. Items that haven't been indexed have the status field set to false. The `errorMessage` and `statusCode` properties indicate the reason for the indexing error:

JSON

```
{  
  "value": [  
    {  
      "key": "unique_key_of_document_1",  
      "status": false,  
      "errorMessage": "The search service is too busy to process this document.  
Please try again later.",  
      "statusCode": 503  
    },  
    {  
      "key": "unique_key_of_document_2",  
      "status": false,  
      "errorMessage": "Document not found.",  
      "statusCode": 404  
    },  
    {  
      "key": "unique_key_of_document_3",  
      "status": false,  
      "errorMessage": "Index is temporarily unavailable because it was updated  
with the 'allowIndexDowntime' flag set to 'true'. Please try again later.",  
      "statusCode": 422  
    }  
  ]  
}
```

The `errorMessage` property indicates the reason for the indexing error if possible.

The following table explains the various per-document status codes that can be returned in the response. Some status codes indicate problems with the request itself, while others indicate temporary error conditions. The latter you should retry after a delay.

[] Expand table

Status code	Meaning	Retryable	Notes
200	Document was successfully modified or deleted.	n/a	Delete operations are idempotent. That is, even if a document key doesn't exist in the index, attempting a delete operation with that key results in a 200 status code.
201	Document was successfully created.	n/a	

Status code	Meaning	Retryable	Notes
400	There was an error in the document that prevented it from being indexed.	No	The error message in the response indicates what is wrong with the document.
404	The document couldn't be merged because the given key doesn't exist in the index.	No	This error doesn't occur for uploads since they create new documents, and it doesn't occur for deletes because they're idempotent.
409	A version conflict was detected when attempting to index a document.	Yes	This can happen when you're trying to index the same document more than once concurrently.
422	The index is temporarily unavailable because it was updated with the 'allowIndexDowntime' flag set to 'true'.	Yes	
429	Too Many Requests	Yes	If you get this error code during indexing, it usually means that you're running low on storage. As you near storage limits , the service can enter a state where you can't add or update until you delete some documents. For more information, see Plan and manage capacity if you want more storage, or free up space by deleting documents.
503	Your search service is temporarily unavailable, possibly due to heavy load.	Yes	Your code should wait before retrying in this case or you risk prolonging the service unavailability.

If your client code frequently encounters a 207 response, one possible reason is that the system is under load. You can confirm this by checking the statusCode property for 503. If the statusCode is 503, we recommend throttling indexing requests. Otherwise, if indexing traffic doesn't subside, the system could start rejecting all requests with 503 errors.

Status code 429 indicates that you've exceeded your quota on the number of documents per index. You must either [upgrade for higher capacity limits](#) or create a new index.

(!) Note

When you upload `DateTimeOffset` values with time zone information to your index, Azure AI Search normalizes these values to UTC. For example, 2024-01-13T14:03:00-08:00 is

stored as 2024-01-13T22:03:00Z. If you need to store time zone information, add an extra column to your index for this data point.

Tips for incremental indexing

- [Indexers automate incremental indexing](#). If you can use an indexer, and if the data source supports change tracking, you can run the indexer on a recurring schedule to add, update, or overwrite searchable content so that it's synchronized to your external data.
- If you're making index calls directly through the [push API](#), use `mergeOrUpload` as the search action.
- The payload must include the keys or identifiers of every document you want to add, update, or delete.
- If your index includes vector fields and you set the [stored property to false](#), make sure you provide the vector in your partial document update, even if the value is unchanged. A side effect of setting `stored` to false is that vectors are dropped on a reindexing operation. Providing the vector in the documents payload prevents this from happening.
- To update the contents of simple fields and subfields in complex types, list only the fields you want to change. For example, if you only need to update a description field, the payload should consist of the document key and the modified description. Omitting other fields retains their existing values.
- To merge inline changes into string collection, provide the entire value. Recall the `tags` field example from the previous section. New values overwrite the old values for an entire field, and there's no merging within the content of a field.

Here's a [REST API example](#) demonstrating these tips:

```
rest
```

```
### Get Stay-Kay City Hotel by ID
GET {{baseUrl}}/indexes/hotels-vector-quickstart/docs('1')?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


### Change the description, city, and tags for Stay-Kay City Hotel
POST {{baseUrl}}/indexes/hotels-vector-quickstart/docs/search.index?api-
version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
```

```

    "value": [
      {
        "@search.action": "mergeOrUpload",
        "HotelId": "1",
        "Description": "I'm overwriting the description for Stay-Kay City
Hotel.",
        "Tags": ["my old item", "my new item"],
        "Address": {
          "City": "Gotham City"
        }
      }
    ]
}

### Retrieve the same document, confirm the overwrites and retention of all other
values
GET {{baseUrl}}/indexes/hotels-vector-quickstart/docs('1')?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

```

Update an index schema

The index schema defines the physical data structures created on the search service, so there aren't many schema changes that you can make without incurring a full rebuild.

Updates with no rebuild

The following list enumerates the schema changes that can be introduced seamlessly into an existing index. Generally, the list includes new fields and functionality used during query execution.

- Add an [index description \(preview\)](#)
- Add a new field
- Set the `retrievable` attribute on an existing field
- Update `searchAnalyzer` on a field having an existing `indexAnalyzer`
- Add a new [analyzer definition](#) in an index (which can be applied to new fields)
- Add, update, or delete [scoring profiles](#)
- Add, update, or delete [synonymMaps](#)
- Add, update, or delete [semantic configurations](#)
- Add, update, or delete CORS settings

The order of operations is:

1. [Get the index definition.](#)

2. Revise the schema with updates from the previous list.

3. [Update index schema](#) on the search service.

4. [Update index content](#) to match your revised schema if you added a new field. For all other changes, the existing indexed content is used as-is.

When you update an index schema to include a new field, existing documents in the index are given a null value for that field. On the next indexing job, values from external source data replace the nulls added by Azure AI Search.

There should be no query disruptions during the updates, but query results will vary as the updates take effect.

Updates requiring a rebuild

Some modifications require an index drop and rebuild, replacing a current index with a new one.

[+] [Expand table](#)

Action	Description
Delete a field	To physically remove all traces of a field, you have to rebuild the index. When an immediate rebuild isn't practical, you can modify application code to redirect access away from an obsolete field or use the searchFields and select query parameters to choose which fields are searched and returned. Physically, the field definition and contents remain in the index until the next rebuild, when you apply a schema that omits the field in question.
Change a field definition	Revisions to a field name, data type, or specific index attributes (searchable, filterable, sortable, facetable) require a full rebuild.
Assign an analyzer to a field	Analyzers are defined in an index, assigned to fields, and then invoked during indexing to inform how tokens are created. You can add a new analyzer definition to an index at any time, but you can only <i>assign</i> an analyzer when the field is created. This is true for both the analyzer and indexAnalyzer properties. The searchAnalyzer property is an exception (you can assign this property to an existing field).
Update or delete an analyzer definition in an index	You can't delete or change an existing analyzer configuration (analyzer, tokenizer, token filter, or char filter) in the index unless you rebuild the entire index.
Add a field to a suggester	If a field already exists and you want to add it to a Suggesters construct, rebuild the index.

Action	Description
Upgrade your service or tier	If you need more capacity, check if you can upgrade your service or switch to a higher pricing tier . If not, you must create a new service and rebuild your indexes from scratch. To help automate this process, you can use a code sample that backs up your index to a series of JSON files. You can then recreate the index in a search service you specify.

The order of operations is:

1. [Get an index definition](#) in case you need it for future reference, or to use as the basis for a new version.
2. Consider using a backup and restore solution to preserve a copy of index content. There are solutions in [C#](#) and in [Python](#). We recommend the Python version because it's more up to date.
If you have capacity on your search service, keep the existing index while creating and testing the new one.
3. [Drop the existing index](#). Queries targeting the index are immediately dropped. Remember that deleting an index is irreversible, destroying physical storage for the fields collection and other constructs.
4. [Post a revised index](#), where the body of the request includes changed or modified field definitions and configurations.
5. [Load the index with documents](#) from an external source. Documents are indexed using the field definitions and configurations of the new schema.

When you create the index, physical storage is allocated for each field in the index schema, with an inverted index created for each searchable field and a vector index created for each vector field. Fields that aren't searchable can be used in filters or expressions, but don't have inverted indexes and aren't full-text or fuzzy searchable. On an index rebuild, these inverted indexes and vector indexes are deleted and recreated based on the index schema you provide.

To minimize disruption to application code, consider [creating an index alias](#). Application code references the alias, but you can update the name of the index that the alias points to.

Add an index description

An index has a `description` property that you can specify and use when a system must access several indexes and make a decision based on the description. Consider a Model Context

Protocol (MCP) server that must pick the correct index at run time. The decision can be based on the description rather than on the index name alone.

An index description is a schema update, and you can add it without having to rebuild the entire index.

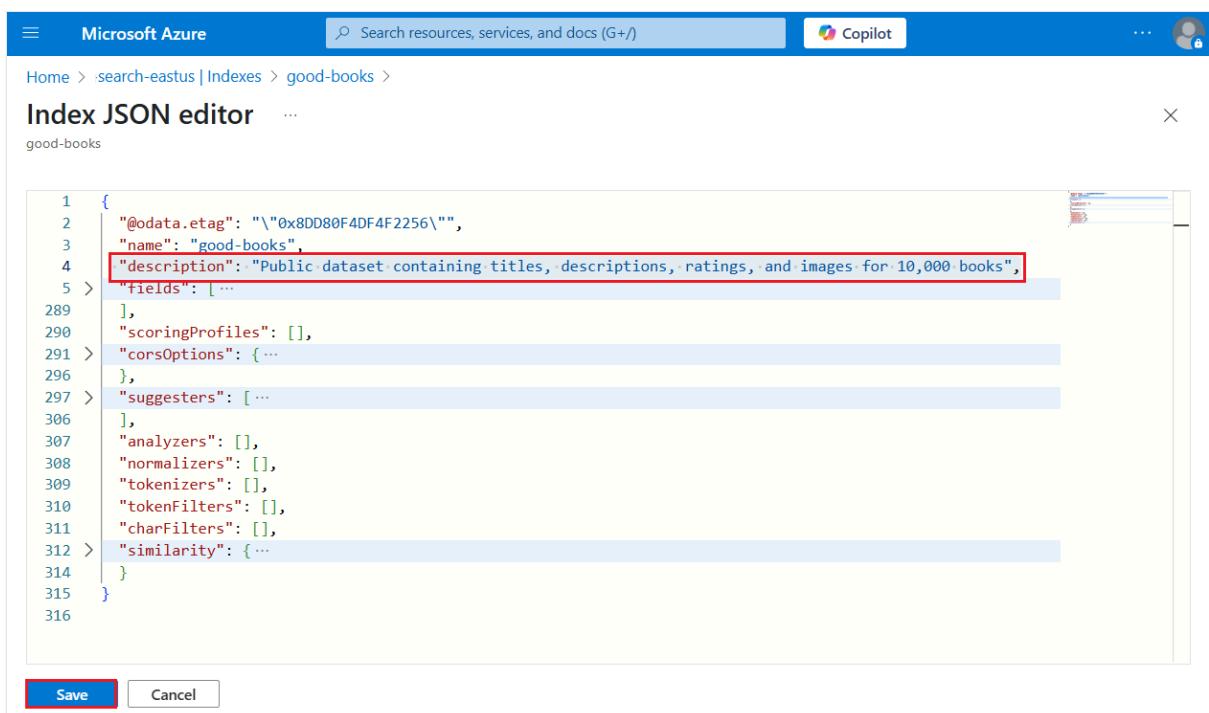
- String length is 4,000 characters maximum.
- Content must be human-readable, in Unicode. Your use-case should determine which language to use.

Support for an index description is provided in the preview REST API, the Azure portal, or in a prerelease Azure SDK package that provides the feature.

Azure portal

The Azure portal supports the latest preview API.

1. Sign in to the Azure portal and find your search service.
2. Under **Search management > Indexes**, select an index.
3. Select **Edit JSON**.
4. Insert `"description"`, followed by the description. The value must be less than 4,000 characters and in Unicode.



The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header bar with the Microsoft Azure logo, a search bar, and a Copilot button. Below the header, the URL shows the user is in the 'search-eastus | Indexes' section, specifically viewing the 'good-books' index. The main area is titled 'Index JSON editor' with the index name 'good-books' below it. The JSON code for the index is displayed in a code editor. A specific line of code, '4 > "description": "Public dataset containing titles, descriptions, ratings, and images for 10,000 books",', is highlighted with a red rectangular box. At the bottom of the code editor, there are two buttons: a blue 'Save' button and a white 'Cancel' button.

```
1  {
2    "@odata.etag": "\"0x8DD80F4DF4F2256\"",
3    "name": "good-books",
4    "description": "Public dataset containing titles, descriptions, ratings, and images for 10,000 books",
5    "fields": [
289    ],
290    "scoringProfiles": [],
291    "corsOptions": {...},
296    },
297    "suggesters": [...],
306    ],
307    "analyzers": [],
308    "normalizers": [],
309    "tokenizers": [],
310    "tokenFilters": [],
311    "charFilters": [],
312    "similarity": {...}
314  }
315}
316
```

5. Save the index.

Balancing workloads

Indexing doesn't run in the background, but the search service will balance any indexing jobs against ongoing queries. During indexing, you can [monitor query requests](#) in the Azure portal to ensure queries are completing in a timely manner.

If indexing workloads introduce unacceptable levels of query latency, conduct [performance analysis](#) and review these [performance tips](#) for potential mitigation.

Check for updates

You can begin querying an index as soon as the first document is loaded. If you know a document's ID, the [Lookup Document REST API](#) returns the specific document. For broader testing, you should wait until the index is fully loaded, and then use queries to verify the context you expect to see.

You can use [Search Explorer](#) or a [REST client](#) to check for updated content.

If you added or renamed a field, use `select` to return that field:

JSON

```
"search": "*",
"select": "document-id, my-new-field, some-old-field",
"count": true
```

The Azure portal provides index size and vector index size. You can check these values after updating an index, but remember to expect a small delay as the service processes the change and to account for portal refresh rates, which can be a few minutes.

Delete orphan documents

Azure AI Search supports document-level operations so that you can look up, update, and delete a specific document in isolation. The following example shows how to delete a document.

Deleting a document doesn't immediately free up space in the index. Every few minutes, a background process performs the physical deletion. Whether you use the Azure portal or an API to return index statistics, you can expect a small delay before the deletion is reflected in the Azure portal and API metrics.

1. Identify which field is the document key. In the Azure portal, you can view the fields of each index. Document keys are string fields and are denoted with a key icon to make

them easier to spot.

2. Check the values of the document key field: `search=*&$select=HotelId`. A simple string is straightforward, but if the index uses a base-64 encoded field, or if search documents were generated from a `parsingMode` setting, you might be working with values that you aren't familiar with.
3. [Look up the document](#) to verify the value of the document ID and to review its content before deleting it. Specify the key or document ID in the request. The following examples illustrate a simple string for the [Hotels sample index](#) and a base-64 encoded string for the `metadata_storage_path` key of the [cog-search-demo index](#).

HTTP

```
GET https://[service name].search.windows.net/indexes/hotel-sample-index/docs/1111?api-version=2025-09-01
```

HTTP

```
GET https://[service name].search.windows.net/indexes/cog-search-demo/docs/aHR0cHM6Ly9oZWlkaWJsb2JzdG9yYWdlMi5ibG9iLmNvcnUud2luZG93cy5uZXQvY29nLXN1YXJjaC1kZW1vL2d1dGhyawUuanBn0?api-version=2025-09-01
```

4. [Delete the document](#) using a delete `@search.action` to remove it from the search index.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/index?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]
{
  "value": [
    {
      "@search.action": "delete",
      "id": "1111"
    }
  ]
}
```

See also

- [Indexer overview](#)
- [Index large data sets at scale](#)
- [Indexing in the Azure portal](#)

- [Azure SQL Database indexer](#)
- [Azure Cosmos DB for NoSQL indexer](#)
- [Azure blob indexer](#)
- [Azure tables indexer](#)
- [Security in Azure AI Search](#)

Manage an index in Azure AI Search

07/03/2025

After you [create an index](#), you can use the [Azure portal](#) to access its statistics and definition or remove it from your search service.

This article describes how to manage an index without affecting its content. For guidance on modifying an index definition, see [Update or rebuild an index in Azure AI Search](#).

Limitations

The pricing tier of your search service determines the maximum number and size of your indexes, fields, and documents. For more information, see [Service limits in Azure AI Search](#).

Otherwise, the following limitations apply to index management:

- You can't take an index offline for maintenance. Indexes are always available for search operations.
- You can't directly copy or duplicate an index within or across search services. However, you can use the backup and restore sample for [.NET](#) or [Python](#) to achieve similar functionality.

View all indexes

To view all your indexes:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Search management > Indexes**.

Name	Document count	Vector index quota usage	Total storage size
index-1	50	0 Bytes	578.66 KB
index-2	7	122.58 KB	454.76 KB
index-3	100	408.11 KB	1.73 MB
index-4	100	408.26 KB	1.73 MB
index-5	105	1.24 MB	4.5 MB
index-6	20	242.36 KB	919.33 KB

By default, the indexes are sorted by name in ascending order. You can sort by **Name**, **Document count**, **Vector index quota usage**, or **Total storage size** by selecting the corresponding column header.

View an index's statistics

On the index page, the portal provides the following statistics:

- Number of documents in the index.
- Storage space used by the index.
- Vector storage space used by the index.
- Maximum storage space for each index on your search service, which [depends on your pricing tier](#). This value doesn't represent the total storage currently available to the index.

Documents	50	Total storage	578.66 KB	Vector index quota usage	0 Bytes	Max storage	15 GB
-----------	----	---------------	-----------	--------------------------	---------	-------------	-------

View an index's definition

Each index is defined by fields and optional components that enhance search capabilities, such as analyzers, normalizers, tokenizers, and synonym maps. This definition determines the index's structure and behavior during indexing and querying.

On the index page, select **Edit JSON** to view its complete definition.

The screenshot shows the Azure portal interface for managing an index named 'my-index'. At the top, there are navigation links: 'Home > my-search-service | Indexes > my-index ...'. Below this is a toolbar with icons for 'Save', 'Discard', 'Refresh', 'Create demo app', 'Edit JSON' (which is highlighted with a red box), 'Delete', and 'Encryption'. Key statistics are displayed: 'Documents 50', 'Total storage 578.66 KB', 'Vector index quota usage 0 Bytes', and 'Max storage 15 GB'. A horizontal menu bar includes 'Search explorer' (underlined), 'Fields', 'CORS', 'Scoring profiles', 'Semantic configurations', and 'Vector profiles'. Below the menu is a search bar with a 'Search' button and a magnifying glass icon. The 'Results' section shows a single item with the number '1'.

Delete an index

⚠ Warning

You can't undo an index deletion. Before you proceed, make sure that you want to permanently remove the index and its documents from your search service.

On the index page, select **Delete** to initiate the deletion process.

This screenshot is identical to the one above, showing the 'my-index' index details page. However, the 'Delete' button in the toolbar is now highlighted with a red box, indicating it is the selected action.

The portal prompts you to confirm the deletion. After you select **Delete**, check your notifications to confirm that the deletion was successful.

Delete confirmation

This action will permanently delete index 'my-index' and all its data.

Delete

Cancel



Related content

- [Search indexes in Azure AI Search](#)
- [Create an index](#)
- [Load data into an index](#)
- [Update or rebuild an index](#)

Create an index alias in Azure AI Search

10/03/2025

ⓘ Important

Index aliases are currently in public preview and available under [supplemental terms of use](#).

In Azure AI Search, an index alias is a secondary name for a search index. You can create an alias that maps to a search index and substitute the alias name in places where you would otherwise reference an index name. This gives you flexibility if you ever need to change which index your application is pointing to. Instead of updating the references to the index name in your production code, you can just update the mapping for your alias.

You can create and manage aliases in Azure AI Search service via HTTP requests (POST, GET, PUT, DELETE) against a given alias resource. Aliases are service level resources and maintained independently from search indexes. Once a search index is created, you can create an alias that maps to that search index.

Before using an alias, your application sends requests directly to `hotel-samples-index`.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2025-08-01-preview
{
  "search": "pool spa +airport",
  "select": "HotelId, HotelName, Category, Description",
  "count": true
}
```

After using an alias, your application sends requests to `my-alias`, which maps to `hotel-samples-index`.

HTTP

```
POST /indexes/my-alias/docs/search?api-version=2025-08-01-preview
{
  "search": "pool spa +airport",
  "select": "HotelId, HotelName, Category, Description",
  "count": true
}
```

Supported scenarios

You can only use an alias with document operations or to get and update an index definition.

Aliases can't be used to [delete an index](#), or [test text tokenization](#), or be referenced as the `targetIndexName` on an [indexer](#) or [knowledge source](#).

Create an index alias

Creating an alias establishes a mapping between an alias name and an index name. If the request is successful, the alias can be used for indexing, querying, and other operations.

Updating an alias allows you to map that alias to a different search index. When you update an existing alias, the entire definition is replaced with the contents of the request body. In general, the best pattern to use for updates is to retrieve the alias definition with a GET, modify it, and then update it with PUT.

You can create an alias using the preview REST API, the preview SDKs, or through the [Azure portal](#). An alias consists of the `name` of the alias and the name of the search index that the alias is mapped to. Only one index name can be specified in the `indexes` array.

The maximum number of aliases that you can create varies by pricing tier. For more information, see [Service limits](#).

REST API

You can use the [Create or Update Alias \(REST preview\)](#) to create an index alias.

HTTP

```
POST /aliases?api-version=2025-08-01-preview
{
    "name": "my-alias",
    "indexes": ["hotel-samples-index"]
}
```

Send requests to an index alias

Aliases can be used for all document operations including querying, indexing, suggestions, and autocomplete.

This query sends the request to `my-alias`, which is mapped to an actual index on your search service.

HTTP

```
POST /indexes/my-alias/docs/search?api-version=2025-08-01-preview
{
    "search": "pool spa +airport",
    "searchMode": any,
    "queryType": "simple",
    "select": "HotelId, HotelName, Category, Description",
    "count": true
}
```

Get an alias definition

This request returns a list of existing alias objects by name.

HTTP

```
GET https://[service name].search.windows.net/aliases?api-version=[api-
version]&$select=name
api-key: [admin key]
```

This request returns an alias definition

HTTP

```
GET https://[service name].search.windows.net/aliases/my-alias?api-version=[api-
version]
api-key: [admin key]
```

Update an alias

The most common update to an alias is changing the index name when the underlying index is replaced with a newer version.

PUT is required for alias updates as described in [Create or Update Alias \(REST preview\)](#).

HTTP

```
PUT /aliases/my-alias?api-version=2025-08-01-preview
{
    "name": "my-alias",
```

```
    "indexes": ["hotel-samples-index2"]  
}
```

An update to an alias may take up to 10 seconds to propagate through the system so you should wait at least 10 seconds before deleting the index that the alias was previously mapped to.

If you attempt to delete an index that is currently mapped to an alias, the operation will fail with 400 (Bad Request) and an error message stating that the alias(es) that's mapped to that index must be deleted or mapped to a different index before the index can be deleted.

See also

- [Drop and rebuild an index in Azure AI Search](#)

Index large data sets in Azure AI Search

10/06/2025

If you need to index large or complex data sets in your search solution, this article explores strategies to accommodate long-running processes on Azure AI Search.

These strategies assume familiarity with the [two basic approaches for importing data](#): *pushing* data into an index, or *pulling* in data from a supported data source using a [search indexer](#). If your scenario involves computationally intensive [AI enrichment](#), then indexers are required, given the skillset dependency on indexers.

This article complements [Tips for better performance](#), which offers best practices on index and query design. A well-designed index that includes only the fields and attributes you need is an important prerequisite for large-scale indexing.

We recommend using a search service created after April 3, 2024 for [higher storage per partition](#). Older services can also be [upgraded to benefit from higher partition storage](#).

(!) Note

The strategies described in this article assume a single large data source. If your solution requires indexing from multiple data sources, see [Index multiple data sources in Azure AI Search](#) for a recommended approach.

Index data using the push APIs

Push APIs, such as the [Documents Index REST API](#) or the [IndexDocuments method \(Azure SDK for .NET\)](#), are the most prevalent form of indexing in Azure AI Search. For solutions that use a push API, the strategy for long-running indexing has one or both of the following components:

- Batching documents
- Managing threads

Batch multiple documents per request

A simple mechanism for indexing a large quantity of data is to submit multiple documents or records in a single request. As long as the entire payload is under 16 MB, a request can handle up to 1,000 documents in a bulk upload operation. These limits apply whether you're using the [Documents Index REST API](#) or the [IndexDocuments method](#) in the .NET SDK. Using either API, you can package 1,000 documents in the body of each request.

Batching documents significantly shortens the amount of time it takes to work through a large data volume. Determining the optimal batch size for your data is a key component of optimizing indexing speeds. The two primary factors influencing the optimal batch size are:

- The schema of your index
- The size of your data

Because the optimal batch size depends on your index and your data, the best approach is to test different batch sizes to determine which one results in the fastest indexing speeds for your scenario. For sample code to test batch sizes using the .NET SDK, see [Tutorial: Optimize indexing with the push API](#).

Manage threads and a retry strategy

Indexers have built-in thread management, but when you're using the push APIs, your application code needs to manage threads. Make sure there are sufficient threads to make full use of the available capacity, especially if you recently [upgraded your service](#), [switched to a higher pricing tier](#), or [increased partitions](#).

1. [Increase the number of concurrent threads](#) in your client code.
2. As you ramp up the requests hitting the search service, you might encounter [HTTP status codes](#) indicating the request didn't fully succeed. During indexing, two common HTTP status codes are:
 - **503 Service Unavailable:** This error means that the system is under heavy load and your request can't be processed at this time.
 - **207 Multi-Status:** This error means that some documents succeeded, but at least one failed.
3. To handle failures, requests should be retried using an [exponential backoff retry strategy](#).

The Azure .NET SDK automatically retries 503s and other failed requests, but you need to implement your own logic to retry 207s. Open-source tools such as [Polly](#) can also be used to implement a retry strategy.

Use indexers and the pull APIs

[Indexers](#) have several capabilities that are useful for long-running processes:

- Batching documents
- Parallel indexing over partitioned data

- Scheduling and change detection for indexing only new and changed documents over time

Indexer schedules can resume processing at the last known stopping point. If data isn't fully indexed within the processing window, the indexer picks up wherever it left off on the next run, assuming you're using a data source that provides change detection.

Partitioning data into smaller individual data sources enables parallel processing. You can break up source data, such as into multiple containers in Azure Blob Storage, [create a data source](#) for each partition, and then [run the indexers in parallel](#), subject to the number of search units of your search service.

Check indexer batch size

As with the push API, indexers allow you to configure the number of items per batch. For indexers based on the [Create Indexer REST API](#), you can set the `batchSize` argument to customize this setting to better match the characteristics of your data.

Default batch sizes are data-source specific. Azure SQL Database and Azure Cosmos DB have a default batch size of 1,000. In contrast, Azure Blob and SharePoint Online (Preview) indexing sets batch size at 10 documents in recognition of the larger average document size.

Schedule indexers for long-running processes

Indexer scheduling is an important mechanism for processing large data sets and for accommodating slow-running processes like image analysis in an enrichment pipeline.

Typically, indexer processing runs within a two-hour window. If the indexing workload takes days rather than hours to complete, you can put the indexer on a consecutive, recurring schedule that starts every two hours. Assuming the data source has [change tracking enabled](#), the indexer resumes processing where it last left off. At this cadence, an indexer can work its way through a document backlog over a series of days until all unprocessed documents are processed. This pattern is especially important during the initial run or when indexing large blob containers, where the blob listing phase alone can take multiple hours or days. During this time, the indexer would show no blobs being processed, but unless an error is reported, it is likely still iterating through the blob list. Document processing and enrichment begin only after this phase completes, and this behavior is expected.

JSON

```
{  
  "dataSourceName" : "hotels-ds",  
  "targetIndexName" : "hotels-idx",
```

```
        "schedule" : { "interval" : "PT2H", "startTime" : "2024-01-01T00:00:00Z" }  
    }
```

When there are no longer any new or updated documents in the data source, indexer execution history reports 0/0 documents processed, and no processing occurs.

For more information about setting schedules, see [Create Indexer REST API](#) or see [Schedule indexers for Azure AI Search](#).

ⓘ Note

Some indexers that run on an older runtime architecture have a 24-hour rather than 2-hour maximum processing window. The two-hour limit is for newer content processors that run in an [internally managed multitenant environment](#). Whenever possible, Azure AI Search tries to offload indexer and skillset processing to the multitenant environment. If the indexer can't be migrated, it runs in the private environment and it can run for as long as 24 hours. If you're scheduling an indexer that exhibits these characteristics, assume a 24-hour processing window.

Run indexers in parallel

If you partition your data, you can create multiple indexer-data-source combinations that pull from each data source and write to the same search index. Because each indexer is distinct, you can run them at the same time, populating a search index more quickly than if you ran them sequentially.

Make sure you have sufficient capacity. One search unit in your service can run one indexer at any given time. Creating multiple indexers is only useful if they can run in parallel.

The number of indexing jobs that can run simultaneously varies for text-based and skills-based indexing. For more information, see [Indexer execution](#).

If your data source is an [Azure Blob Storage container](#) or [Azure Data Lake Storage Gen 2](#), enumerating a large number of blobs can take a long time (even hours) until this operation is completed. As a result, your indexer's *documents succeeded* count doesn't appear to increase during that time and it might seem it's not making any progress, when it is. If you would like document processing to go faster for a large number of blobs, consider partitioning your data into multiple containers and create parallel indexers pointing to a single index.

1. Sign in to the [Azure portal](#) and check the number of search units used by your search service. Select **Settings > Scale** to view the number at the top of the page. The number of indexers that run in parallel is approximately equal to the number of search units.

2. Partition source data among multiple containers or multiple virtual folders inside the same container.
3. Create multiple [data sources](#), one for each partition, paired to its own [indexer](#).
4. Specify the same target search index in each indexer.
5. Schedule the indexers.
6. Review indexer status and execution history for confirmation.

There are some risks associated with parallel indexing. First, recall that indexing doesn't run in the background, increasing the likelihood that queries are throttled or dropped.

Second, Azure AI Search doesn't lock the index for updates. Concurrent writes are managed, invoking a retry if a particular write doesn't succeed on first attempt, but you might notice an increase in indexing failures.

Although multiple indexer-data-source sets can target the same index, be careful of indexer runs that can overwrite existing values in the index. If a second indexer-data-source targets the same documents and fields, any values from the first run are overwritten. Field values are replaced in full; an indexer can't merge values from multiple runs into the same field.

Index big data on Spark

If you have a big data architecture and your data is on a Spark cluster, we recommend [SynapseML for loading and indexing data](#). The tutorial includes steps for calling Azure AI services for AI enrichment, but you can also use the `AzureSearchWriter` API for text indexing.

Related content

- [Tutorial: Optimize indexing by using the push API](#)
- [Tutorial: Index large data from Apache Spark using SynapseML and Azure AI Search](#)
- [Tips for better performance in Azure AI Search](#)
- [Analyze performance in Azure AI Search](#)
- [Indexers in Azure AI Search](#)
- [Monitor indexer status and results in Azure AI Search](#)

Add synonyms in Azure AI Search

Article • 04/14/2025

On a search service, a synonym map associates equivalent terms, expanding the scope of a query without the user having to actually provide the term. For example, assuming *dog*, *canine*, and *puppy* are mapped synonyms, a query on *canine* matches on a document containing *dog*. You might create multiple synonym maps for different languages, such as English and French versions, or lexicons if your content includes technical jargon, slang, or obscure terminology.

Some key points about synonym maps:

- A synonym map is a top-level resource that can be created once and used by many indexes.
- A synonym map applies to string fields.
- You can create and assign a synonym map at any time with no disruption to indexing or queries.
- Your [service tier](#) sets the limits on how many synonym maps you can create.
- Your search service can have multiple synonym maps, but within an index, a field definition can only have one synonym map assignment.

Create a synonym map

A synonym map consists of name, format, and rules that function as synonym map entries. The only format that's supported is `solr`, and the `solr` format determines rule construction.

To create a synonym map, do so programmatically. the Azure portal doesn't support synonym map definitions.

REST

Use the [Create Synonym Map \(REST API\)](#) to create a synonym map.

HTTP

```
POST /synonymmaps?api-version=2024-07-01
{
    "name": "geo-synonyms",
    "format": "solr",
    "synonyms": "
        USA, United States, United States of America\n
        Washington, Wash., WA => WA\n"
}
```

Define rules

Mapping rules adhere to the open-source synonym filter specification of Apache Solr, described in this document: [SynonymGraphFilter](#). The `solr` format supports two kinds of rules:

- equivalency (where terms are equal substitutes in the query)
- explicit mappings (where terms are mapped to one explicit term)

Each rule is delimited by the new line character (`\n`). You can define up to 5,000 rules per synonym map in a free service and 20,000 rules per map in other tiers. Each rule can have up to 20 expansions, or items in a rule. For more information, see [Synonym limits](#).

Query parsers automatically lower-case any upper or mixed case terms. To preserve special characters in the string, such as a comma or dash, add the appropriate escape characters when creating the synonym map.

Equivalency rules

Rules for equivalent terms are comma-delimited within the same rule. In the first example, a query on *USA* expands to *USA* OR *"United States"* OR *"United States of America."* Notice that if you want to match on a phrase, the query itself must be a quote-enclosed phrase query.

In the equivalence case, a query for *dog* expands the query to also include *puppy* and *canine*.

JSON

```
{  
  "format": "solr",  
  "synonyms": "  
    USA, United States, United States of America\n    dog, puppy, canine\n    coffee, latte, cup of joe, java\n"}  
}
```

Explicit mapping

Rules for an explicit mapping are denoted by an arrow `=>`. When specified, a term sequence of a search query that matches the left-hand side of `=>` is replaced with the alternatives on the right-hand side at query time.

In the explicit case, a query for *Washington*, *Wash.* or *WA* is rewritten as *WA*, and the query engine only looks for matches on the term *WA*. Explicit mapping only applies in the direction

specified, and doesn't rewrite the query *WA* to *Washington* in this case.

JSON

```
{  
  "format": "solr",  
  "synonyms": "  
    Washington, Wash., WA => WA\\n  
    California, Calif., CA => CA\\n"  
}
```

Escaping special characters

Synonyms are analyzed during query processing just like any other query term, which means that rules for reserved and special characters apply to the terms in your synonym map. The list of characters that require escaping varies between the simple syntax and full syntax:

- **simple syntax** + | " () ' \
- **full syntax** + - & | ! () { } [] ^ " ~ * ? : \ /

To preserve characters that the default analyzer discards, substitute an analyzer that preserves them. Some choices include Microsoft natural [language analyzers](#), which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see [Partial terms, patterns, and special characters](#).

The following example shows an example of how to escape a character with a backslash:

JSON

```
{  
  "format": "solr",  
  "synonyms": "WA\\, USA, WA, Washington\\n"  
}
```

Since the backslash is itself a special character in other languages like JSON and C#, you probably need to double-escape it. Here's an example in JSON:

JSON

```
{  
  "format": "solr",  
  "synonyms": "WA\\\\, USA, WA, Washington"  
}
```

Manage synonym maps

You can update a synonym map without disrupting query and indexing workloads. However, once you add a synonym map to a field, if you then delete a synonym map, any query that includes the fields in question fails with a 404 error.

Creating, updating, and deleting a synonym map is always a whole-document operation. You can't update or delete parts of the synonym map incrementally. Updating even a single rule requires a reload.

Assign synonyms to fields

After you create the synonym map, assign it to a field in your index. To assign synonym maps, do so programmatically. the Azure portal doesn't support synonym map field associations.

- A field must be of type `Edm.String` or `Collection(Edm.String)`
- A field must have `"searchable":true`
- A field can have only one synonym map

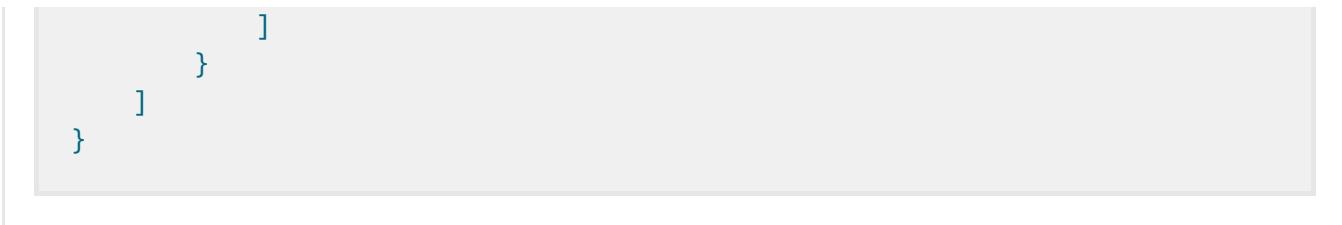
If the synonym map exists on the search service, it's used on the next query, with no reindexing or rebuild required.

REST

Use the [Create or Update Index \(REST API\)](#) to modify a field definition.

HTTP

```
PUT /indexes?api-version=2024-07-01
{
    "name": "hotels-sample-index",
    "fields": [
        {
            "name": "description",
            "type": "Edm.String",
            "searchable": true,
            "synonymMaps": [
                "en-synonyms"
            ]
        },
        {
            "name": "description_fr",
            "type": "Edm.String",
            "searchable": true,
            "analyzer": "fr.microsoft",
            "synonymMaps": [
                "fr-synonyms"
            ]
        }
    ]
}
```



Query on equivalent or mapped fields

A synonym field assignment doesn't change how you write queries. After the synonym map assignment, the only difference is that if a query term exists in the synonym map, the search engine either expands or rewrites the term or phrase, depending on the rule.

How synonyms are used during query execution

Synonyms are a query expansion technique that supplements the contents of an index with equivalent terms, but only for fields that have a synonym assignment. If a field-scoped query *excludes* a synonym-enabled field, you don't see matches from the synonym map.

For synonym-enabled fields, synonyms are subject to the same text analysis as the associated field. For example, if a field is analyzed using the standard Lucene analyzer, synonym terms are also subject to the standard Lucene analyzer at query time. If you want to preserve punctuation, such as periods or dashes, in the synonym term, apply a content-preserving analyzer on the field.

Internally, the synonyms feature rewrites the original query with synonyms by using the OR operator. For this reason, hit highlighting and scoring profiles treat the original term and synonyms as equivalent.

Synonyms apply to free-form text queries only and aren't supported for filters, facets, autocomplete, or suggestions. Autocomplete and suggestions are based only on the original term; synonym matches don't appear in the response.

If you have an existing index in a development (nonproduction) environment, experiment with a small dictionary to see how the addition of synonyms changes the search experience, including impact on scoring profiles, hit highlighting, and suggestions.

Wildcard searches

Synonym expansions don't apply to wildcard search terms; prefix, fuzzy, and regex terms aren't expanded.

If you need to do a single query that applies synonym expansion and wildcard, regex, or fuzzy searches, you can combine the queries using the OR syntax. For example, to combine synonyms with wildcards for simple query syntax, the term would be `<query> | <query>*`.

Next step

[Create a synonym map \(REST API\)](#)

Configure a suggester for autocomplete and suggestions in a query

09/11/2025

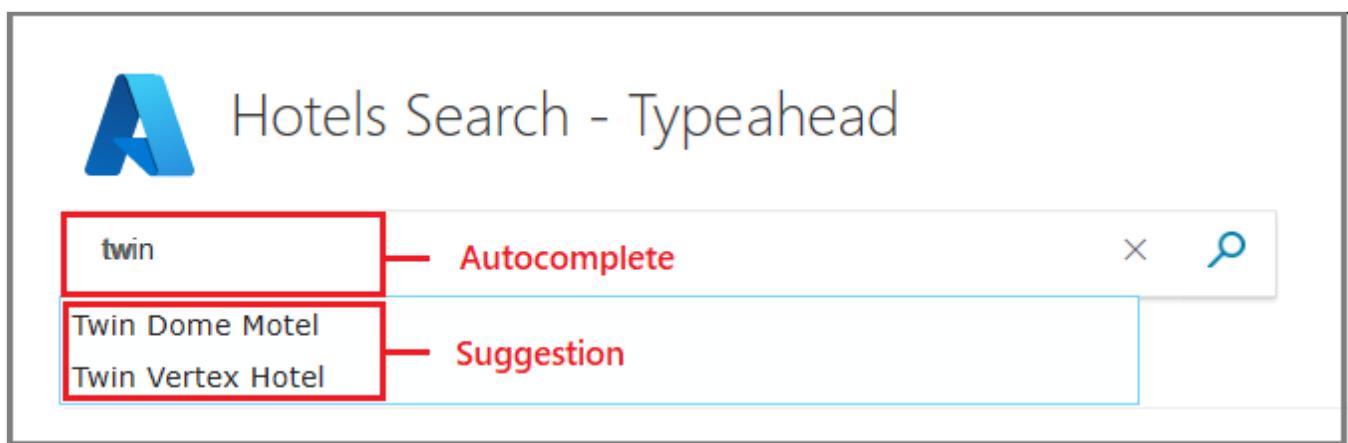
In Azure AI Search, typeahead or "search-as-you-type" is enabled by using a *suggester*. A suggester is a configuration in an index that specifies which fields should be used to populate autocomplete and suggested matches. These fields undergo extra tokenization, generating prefix sequences to support matches on partial terms. For example, a suggester that includes a `city` field with a value for *Seattle* has prefix combinations of *sea*, *seat*, *seatt*, and *seattl* to support typeahead.

Matches on partial terms can be either an autocompleted query or a suggested match. The same suggester supports both experiences.

Typeahead experiences

Typeahead in Azure AI Search can be either *autocomplete*, which completes a partial input for a whole term query, or *suggestions* that invite click through to a particular match. Autocomplete produces a query. Suggestions produce a matching document.

The following screenshot illustrates both. Autocomplete anticipates a potential term, finishing *tw* with *in*. Suggestions are mini search results, where a field like `hotel name` represents a matching hotel search document from the index. For suggestions, you can surface any field that provides descriptive information.



You can use these features separately or together. To implement these behaviors in Azure AI Search, there's an index and query component.

- Add a suggester to a search index definition. The remainder of this article focuses on creating a suggester.

- Call a suggester-enabled query, in the form of a suggestion request or autocomplete request, by using one of the APIs listed in [Use a suggester](#).

Search-as-you-type is enabled on a per-field basis for string fields. You can implement both typeahead behaviors within the same search solution if you want an experience similar to the one indicated in the screenshot. Both requests target the *documents* collection of a specific index, and responses are returned after a user provides at least a three-character input string.

How to create a suggester

To create a suggester, add one to an [index definition](#). A suggester takes a name and a collection of fields over which the typeahead experience is enabled. The best time to create a suggester is when you're also defining the field that uses it.

- Use string fields only.
- If the string field is part of a complex type (for example, a City field within Address), include the parent in the field path: `"Address/City"` (REST, C#, and Python), or `["Address"]["City"]` (JavaScript).
- Use the default standard Lucene analyzer (`"analyzer": null`) or a [language analyzer](#) (for example, `"analyzer": "fr.microsoft"`) on the field.

If you try to create a suggester using preexisting fields, the API disallows it. Prefixes are generated during indexing, when partial terms in two or more character combinations are tokenized alongside whole terms. Given that existing fields are already tokenized, you have to rebuild the index if you want to add them to a suggester. For more information, see [Update or rebuild an index in Azure AI Search](#).

Choose fields

Although a suggester has several properties, it's primarily a collection of string fields for which you're enabling a search-as-you-type experience. There's one suggester for each index, so the suggester list must include all fields that contribute content for both suggestions and autocomplete.

Autocomplete benefits from a larger pool of fields to draw from because the extra content has more term completion potential.

Suggestions, on the other hand, produce better results when your field choice is selective. Remember that the suggestion is a proxy for a search document so pick fields that best represent a single result. Names, titles, or other unique fields that distinguish among multiple

matches work best. If fields consist of repetitive values, the suggestions consist of identical results and a user won't know which one to choose.

To satisfy both search-as-you-type experiences, add all of the fields that you need for autocomplete, but then use `select`, `top`, `filter`, and `searchFields` to control results for suggestions.

Choose analyzers

Your choice of an analyzer determines how fields are tokenized and prefixed. For example, for a hyphenated string like *context-sensitive*, using a language analyzer results in these token combinations: *context*, *sensitive*, *context-sensitive*. Had you used the standard Lucene analyzer, the hyphenated string wouldn't exist.

When evaluating analyzers, consider using the [Analyze Text API](#) for insight into how terms are processed. Once you build an index, you can try various analyzers on a string to view token output.

Fields that use [custom analyzers](#) or [built-in analyzers](#), (except for standard Lucene) are explicitly disallowed to prevent poor outcomes.

! Note

If you need to work around the analyzer constraint, for example if you need a keyword or ngram analyzer for certain query scenarios, you should use two separate fields for the same content. This allows one of the fields to have a suggester, while the other can be set up with a custom analyzer configuration. If you're using an indexer, you can map a source field to two different index fields to support multiple configurations.

Create using the Azure portal

In the Azure portal, you can specify a suggester when you select **Add index**.

1. Select **Add index** and add a string field.
2. Set field attribution to **Searchable**.
3. Select an analyzer.
4. Once fields are defined, select **Autocomplete settings**.
5. Select the searchable string fields for which you want to enable an autocomplete experience.

Create using REST

In the REST API, add suggesters by using [Create Index](#).

JSON

```
{  
  "name": "hotels-sample-index",  
  "fields": [  
    . . .  
    {  
      "name": "HotelName",  
      "type": "Edm.String",  
      "facetable": false,  
      "filterable": false,  
      "key": false,  
      " retrievable": true,  
      "searchable": true,  
      "sortable": false,  
      "analyzer": "en.microsoft",  
      "indexAnalyzer": null,  
      "searchAnalyzer": null,  
      "synonymMaps": [],  
      "fields": []  
    },  
  ],  
  "suggesters": [  
    {  
      "name": "sg",  
      "searchMode": "analyzingInfixMatching",  
      "sourceFields": ["HotelName"]  
    }  
  ],  
  "scoringProfiles": [  
    . . .  
  ]  
}
```

Create using .NET

In C#, define a [SearchSuggester](#) object. `Suggesters` is a collection on a `SearchIndex` object, but it can only take one item. Add a suggester to the index definition.

C#

```
private static void CreateIndex(string indexName, SearchIndexClient indexClient)  
{  
    FieldBuilder fieldBuilder = new FieldBuilder();  
    var searchFields = fieldBuilder.Build(typeof(Hotel));
```

```

var definition = new SearchIndex(indexName, searchFields);

var suggester = new SearchSuggester("sg", new[] { "HotelName", "Category",
"Address/City", "Address/StateProvince" });
definition.Suggesters.Add(suggester);

indexClient.CreateOrUpdateIndex(definition);
}

```

Property reference

[] [Expand table](#)

Property	Description
name	Specified in the suggester definition, but also called on an Autocomplete or Suggestions request.
sourceFields	<p>Specified in the suggester definition. It's a list of one or more fields in the index that are the source of the content for suggestions. Fields must be of type <code>Edm.String</code>. If an analyzer is specified on the field, it must be a named lexical analyzer listed on LexicalAnalyzerName Struct (not a custom analyzer).</p> <p>As a best practice, specify only those fields that lend themselves to an expected and appropriate response, whether it's a completed string in a search bar or a dropdown list.</p> <p>A hotel name is a good candidate because it has precision. Verbose fields like descriptions and comments are too dense. Similarly, repetitive fields, such as categories and tags, are less effective. In the examples, we include <i>category</i> anyway to demonstrate that you can include multiple fields.</p>
searchMode	REST-only parameter, but also visible in the Azure portal. This parameter isn't available in the .NET SDK. It indicates the strategy used to search for candidate phrases. The only mode currently supported is <code>analyzingInfixMatching</code> , which currently matches on the beginning of a term.

Use a suggester

A suggester is used in a query. After a suggester is created, call one of the following APIs for a search-as-you-type experience:

- [Suggest REST API](#)
- [Autocomplete REST API](#)
- [SuggestAsync method](#)
- [AutocompleteAsync method](#)

In a search application, client code should use a library like [jQuery UI Autocomplete](#) to collect the partial query and provide the match. For more information about this task, see [Add autocomplete or suggested results to client code](#).

API usage is illustrated in the following call to the Autocomplete REST API. There are two takeaways from this example. First, as with all queries, the operation is against the documents collection of an index and the query includes a `search` parameter, which in this case provides the partial query. Second, you must add `suggesterName` to the request. If a suggester isn't defined in the index, calls to autocomplete or suggestions fail.

HTTP

```
POST /indexes/myxboxgames/docs/autocomplete?search&api-version=2025-09-01
{
  "search": "minecraf",
  "suggesterName": "sg"
}
```

Sample code

To learn how to use an open source Suggestions package for partial term completion in the client app, see [Explore the .NET search code](#).

Next step

Learn more about request formulation.

[Add autocomplete and search suggestions in client apps](#)

Create an index for multiple languages in Azure AI Search

05/29/2025

If you have strings in multiple languages, you can use [vector search](#) to represent multilingual content mathematically, which is the more modern approach. Alternatively, if you aren't using vectors, you can attach [language analyzers](#) that analyze strings using linguistic rules of a specific language during indexing and query execution. With a language analyzer, you get better handling of diacritics, character variants, punctuation, and word root forms.

Azure AI Search supports Microsoft and Lucene analyzers. By default, the search engine uses Standard Lucene, which is language agnostic. If testing indicates that the default analyzer is insufficient, replace it with a language analyzer.

In Azure AI Search, the two patterns for supporting multiple languages include:

- Create language-specific indexes where all of the human readable content is in the same language, and all searchable string fields are attributed to use the same [language analyzer](#).
- Create a blended index with language-specific versions of each field (for example, `description_en`, `description_fr`, `description_ko`), and then constrain full text search to just those fields at query time. This approach is useful for scenarios where language variants are only needed on a few fields, like a description.

This article focuses on steps and best practices for configuring and querying language-specific fields in a blended index:

- ✓ Define a string field for each language variant.
- ✓ Set a language analyzer on each field.
- ✓ On the query request, set the `searchFields` parameter to specific fields, and then use `select` to return just those fields that have compatible content.

ⓘ Note

If you're using large language models in a retrieval augmented generated (RAG) pattern, you can engineer the prompt to return translated strings. That scenario is out of scope for this article.

Prerequisites

Language analysis applies to fields of type `Edm.String` that are `searchable`, and that contain localized text. If you also need text translation, review the next section to see if AI enrichment meets your needs.

Non-string fields and non-searchable string fields don't undergo lexical analysis and aren't tokenized. Instead, they're stored and returned verbatim.

Add text translation

This article assumes translated strings already exist. If that's not the case, you can attach Azure AI services to an [enrichment pipeline](#), invoking text translation during indexing. Text translation takes a dependency on the indexer feature and Azure AI services, but all setup is done within Azure AI Search.

To add text translation, follow these steps:

1. Verify your content is in a [supported data source](#).
2. [Create a data source](#) that points to your content.
3. [Create a skillset](#) that includes the [Text Translation skill](#).

The Text Translation skill takes a single string as input. If you have multiple fields, can create a skillset that calls Text Translation multiple times, once for each field. Alternatively, you can use the [Text Merger skill](#) to consolidate the content of multiple fields into one long string.

4. Create an index that includes fields for translated strings. Most of this article covers index design and field definitions for indexing and querying multi-language content.
5. [Attach a multi-region Azure AI services resource](#) to your skillset.
6. [Create and run the indexer](#), and then apply the guidance in this article to query just the fields of interest.

💡 Tip

Text translation is built into the [Import data wizard](#). If you have a [supported data source](#) with text you'd like to translate, you can step through the wizard to try out the language detection and translation functionality before writing any code.

Define fields for content in different languages

In Azure AI Search, queries target a single index. Developers who want to provide language-specific strings in a single search experience typically define dedicated fields to store the values: one field for English strings, one for French, and so on.

The `analyzer` property on a field definition is used to set the [language analyzer](#). It's used for both indexing and query execution.

```
JSON

{
  "name": "hotels-sample-index",
  "fields": [
    {
      "name": "Description",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "en.microsoft"
    },
    {
      "name": "Description_fr",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "fr.microsoft"
    }
  ]
}
```

Build and load an index

An intermediate step is [building and populating the index](#) before formulating a query. We mention this step here for completeness. One way to determine index availability is by checking the indexes list in the [portal](#).

Constrain the query and trim results

Parameters on the query are used to limit search to specific fields and then trim the results of any fields not helpful to your scenario.

[] [Expand table](#)

Parameters	Purpose
<code>searchFields</code>	Limits full text search to the list of named fields.

Parameters	Purpose
<code>select</code>	Trims the response to include only the fields you specify. By default, all retrievable fields are returned. The <code>select</code> parameter lets you choose which ones to return.

Given a goal of constraining search to fields containing French strings, you would use `searchFields` to target the query at fields containing strings in that language.

Specifying the analyzer on a query request isn't necessary. A language analyzer on the field definition determines text analysis during query execution. For queries that specify multiple fields, each invoking different language analyzers, the terms or phrases are processed concurrently by the assigned analyzers for each field.

By default, a search returns all fields that are marked as retrievable. As such, you might want to exclude fields that don't conform to the language-specific search experience you want to provide. Specifically, if you limited search to a field with French strings, you probably want to exclude fields with English strings from your results. Using the `select` query parameter gives you control over which fields are returned to the calling application.

Example in REST

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
    "search": "animaux acceptés",
    "searchFields": "Tags, Description_fr",
    "select": "HotelName, Description_fr, Address/City, Address/StateProvince,
    Tags",
    "count": "true"
}
```

Example in C#

C#

```
private static void RunQueries(SearchClient srchclient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    options = new SearchOptions()
    {
        IncludeTotalCount = true,
        Filter = "",
    }
```

```

        OrderBy = { "" }  

    };  
  

    options.Select.Add("HotelId");  

    options.Select.Add("HotelName");  

    options.Select.Add("Description_fr");  

    options.SearchFields.Add("Tags");  

    options.SearchFields.Add("Description_fr");  
  

    response = srchclient.Search<Hotel>("*", options);  

    WriteDocuments(response);
}

```

Boost language-specific fields

Sometimes the language of the agent issuing a query isn't known, in which case the query can be issued against all fields simultaneously. IA preference for results in a certain language can be defined using [scoring profiles](#). In the example below, matches found in the description in French are scored higher relative to matches in other languages:

JSON

```

"scoringProfiles": [  

  {  

    "name": "frenchFirst",  

    "text": {  

      "weights": { "description_fr": 2 }
    }
  }
]

```

You would then include the scoring profile in the search request:

HTTP

```

POST /indexes/hotels/docs/search?api-version=2024-07-01  

{  

  "search": "pets allowed",  

  "searchFields": "Tags, Description_fr",  

  "select": "HotelName, Tags, Description_fr",  

  "scoringProfile": "frenchFirst",  

  "count": "true"
}

```

Next steps

- Add a language analyzer
- How full text search works in Azure AI Search
- Search Documents REST API
- AI enrichment overview
- Skillsets overview

Model complex data types in Azure AI Search

Article • 04/14/2025

External datasets used to populate an Azure AI Search index can come in many shapes. Sometimes they include hierarchical or nested substructures. Examples might include multiple addresses for a single customer, multiple colors and sizes for a single product, multiple authors of a single book, and so on. In modeling terms, you might see these structures referred to as *complex*, *compound*, *composite*, or *aggregate* data types. The term Azure AI Search uses for this concept is **complex type**. In Azure AI Search, complex types are modeled using **complex fields**. A complex field is a field that contains children (subfields) which can be of any data type, including other complex types. This works in a similar way as structured data types in a programming language.

Complex fields represent either a single object in the document, or an array of objects, depending on the data type. Fields of type `Edm.ComplexType` represent single objects, while fields of type `collection(Edm.ComplexType)` represent arrays of objects.

Azure AI Search natively supports complex types and collections. These types allow you to model almost any JSON structure in an Azure AI Search index. In previous versions of Azure AI Search APIs, only flattened row sets could be imported. In the newest version, your index can now more closely correspond to source data. In other words, if your source data has complex types, your index can have complex types also.

To get started, we recommend the [Hotels data set](#), which you can load in the **Import data** wizard in the Azure portal. The wizard detects complex types in the source and suggests an index schema based on the detected structures.

ⓘ Note

Support for complex types became generally available starting in `api-version=2019-05-06`.

If your search solution is built on earlier workarounds of flattened datasets in a collection, you should change your index to include complex types as supported in the newest API version. For more information about upgrading API versions, see [Upgrade to the newest REST API version](#) or [Upgrade to the newest .NET SDK version](#).

Example of a complex structure

The following JSON document is composed of simple fields and complex fields. Complex fields, such as `Address` and `Rooms`, have subfields. `Address` has a single set of values for those subfields, since it's a single object in the document. In contrast, `Rooms` has multiple sets of values for its subfields, one for each object in the collection.

JSON

```
{  
    "HotelId": "1",  
    "HotelName": "Stay-Kay City Hotel",  
    "Description": "Ideally located on the main commercial artery of the city in the heart of New York.",  
    "Tags": ["Free wifi", "on-site parking", "indoor pool", "continental breakfast"],  
    "Address": {  
        "StreetAddress": "677 5th Ave",  
        "City": "New York",  
        "StateProvince": "NY"  
    },  
    "Rooms": [  
        {  
            "Description": "Budget Room, 1 Queen Bed (Cityside)",  
            "RoomNumber": 1105,  
            "BaseRate": 96.99,  
        },  
        {  
            "Description": "Deluxe Room, 2 Double Beds (City View)",  
            "Type": "Deluxe Room",  
            "BaseRate": 150.99,  
        }  
        . . .  
    ]  
}
```

Create complex fields

As with any index definition, you can use the Azure portal, [REST API](#), or [.NET SDK](#) to create a schema that includes complex types.

Other Azure SDKs provide samples in [Python](#), [Java](#), and [JavaScript](#).

Azure portal

1. Sign in to the [Azure portal](#).
2. On the search service **Overview** page, select the **Indexes** tab.

3. Open an existing index or create a new index.
4. Select the **Fields** tab, and then select **Add field**. An empty field is added. If you're working with an existing fields collection, scroll down to set up the field.
5. Give the field a name and set the type to either `Edm.ComplexType` or `Collection(Edm.ComplexType)`.
6. Select the ellipses on the far right, and then select either **Add field** or **Add subfield**, and then assign attributes.

Complex collection limits

During indexing, you can have a maximum of 3,000 elements across all complex collections within a single document. An element of a complex collection is a member of that collection. For Rooms (the only complex collection in the Hotel example), each room is an element. In the example above, if the "Stay-Kay City Hotel" had 500 rooms, the hotel document would have 500 room elements. For nested complex collections, each nested element is also counted, in addition to the outer (parent) element.

This limit applies only to complex collections, and not complex types (like Address) or string collections (like Tags).

Update complex fields

All of the [reindexing rules](#) that apply to fields in general still apply to complex fields. Adding a new field to a complex type doesn't require an index rebuild, but most other modifications do require a rebuild.

Structural updates to the definition

You can add new subfields to a complex field at any time without the need for an index rebuild. For example, adding "ZipCode" to `Address` or "Amenities" to `Rooms` is allowed, just like adding a top-level field to an index. Existing documents have a null value for new fields until you explicitly populate those fields by updating your data.

Notice that within a complex type, each subfield has a type and can have attributes, just as top-level fields do

Data updates

Updating existing documents in an index with the `upload` action works the same way for complex and simple fields: all fields are replaced. However, `merge` (or `mergeOrUpload` when applied to an existing document) doesn't work the same across all fields. Specifically, `merge` doesn't support merging elements within a collection. This limitation exists for collections of primitive types and complex collections. To update a collection, you need to retrieve the full collection value, make changes, and then include the new collection in the Index API request.

Search complex fields in text queries

Free-form search expressions work as expected with complex types. If any searchable field or subfield anywhere in a document matches, then the document itself is a match.

Queries get more nuanced when you have multiple terms and operators, and some terms have field names specified, as is possible with the [Lucene syntax](#). For example, this query attempts to match two terms, "Portland" and "OR", against two subfields of the Address field:

```
search=Address/City:Portland AND Address/State:OR
```

Queries like this are *uncorrelated* for full-text search, unlike filters. In filters, queries over subfields of a complex collection are correlated using range variables in [any or all](#). The Lucene query above returns documents containing both "Portland, Maine" and "Portland, Oregon", along with other cities in Oregon. This happens because each clause applies to all values of its field in the entire document, so there's no concept of a "current subdocument". For more information on this, see [Understanding OData collection filters in Azure AI Search](#).

Search complex fields in RAG queries

A RAG pattern passes search results to a chat model for generative AI and conversational search. By default, search results passed to an LLM are a flattened rowset. However, if your index has complex types, your query can provide those fields if you first convert the search results to JSON, and then pass the JSON to the LLM.

A partial example illustrates the technique:

- Indicate the fields you want in the prompt or in the query
- Make sure the fields are searchable and retrievable in the index
- Select the fields for the search results
- Format the results as JSON
- Send the request for chat completion to the model provider

```

import json

# Query is the question being asked. It's sent to the search engine and the LLM.
query="Can you recommend a few hotels that offer complimentary breakfast? Tell me
their description, address, tags, and the rate for one room they have which sleep
4 people."

# Set up the search results and the chat thread.
# Retrieve the selected fields from the search index related to the question.
selected_fields = ["HotelName", "Description", "Address", "Rooms", "Tags"]
search_results = search_client.search(
    search_text=query,
    top=5,
    select=selected_fields,
    query_type="semantic"
)
sources_filtered = [{field: result[field] for field in selected_fields} for result
in search_results]
sources_formatted = "\n".join([json.dumps(source) for source in sources_filtered])

response = openai_client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": GROUNDED_PROMPT.format(query=query,
sources=sources_formatted)
        }
    ],
    model=AZURE_DEPLOYMENT_MODEL
)

print(response.choices[0].message.content)

```

For the end-to-end example, see [Quickstart: Generative search \(RAG\) with grounding data from Azure AI Search](#).

Select complex fields

The `$select` parameter is used to choose which fields are returned in search results. To use this parameter to select specific subfields of a complex field, include the parent field and subfield separated by a slash (/).

```
$select=HotelName, Address/City, Rooms/BaseRate
```

Fields must be marked as Retrievable in the index if you want them in search results. Only fields marked as Retrievable can be used in a `$select` statement.

Filter, facet, and sort complex fields

The same [OData path syntax](#) used for filtering and fielded searches can also be used for faceting, sorting, and selecting fields in a search request. For complex types, rules apply that govern which subfields can be marked as sortable or facetable. For more information on these rules, see the [Create Index API reference](#).

Faceting subfields

Any subfield can be marked as facetable unless it is of type `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)`.

The document counts returned in the facet results are calculated for the parent document (a hotel), not the subdocuments in a complex collection (rooms). For example, suppose a hotel has 20 rooms of type "suite". Given this facet parameter `facet=Rooms/Type`, the facet count is one for the hotel, not 20 for the rooms.

Sorting complex fields

Sort operations apply to documents (Hotels) and not subdocuments (Rooms). When you have a complex type collection, such as Rooms, it's important to realize that you can't sort on Rooms at all. In fact, you can't sort on any collection.

Sort operations work when fields have a single value per document, whether the field is a simple field, or a subfield in a complex type. For example, `Address/City` is allowed to be sortable because there's only one address per hotel, so `$orderby=Address/City` sorts hotels by city.

Filtering on complex fields

You can refer to subfields of a complex field in a filter expression. Just use the same [OData path syntax](#) that's used for faceting, sorting, and selecting fields. For example, the following filter returns all hotels in Canada:

```
$filter=Address/Country eq 'Canada'
```

To filter on a complex collection field, you can use a **lambda expression** with the [any and all operators](#). In that case, the **range variable** of the lambda expression is an object with subfields. You can refer to those subfields with the standard OData path syntax. For example, the following filter returns all hotels with at least one deluxe room and all nonsmoking rooms:

```
$filter=Rooms/any(room: room/Type eq 'Deluxe Room') and Rooms/all(room: not room/SmokingAllowed)
```

As with top-level simple fields, simple subfields of complex fields can only be included in filters if they have the **filterable** attribute set to `true` in the index definition. For more information, see the [Create Index API reference](#).

Workaround for the complex collection limit

Recall that Azure AI Search limits complex objects in a collection to 3,000 objects per document. Exceeding this limit results in the following message:

```
A collection in your document exceeds the maximum elements across all complex collections limit.  
The document with key '1052' has '4303' objects in collections (JSON arrays).  
At most '3000' objects are allowed to be in collections across the entire document.  
Remove objects from collections and try indexing the document again."
```

If you need more than 3,000 items, you can pipe (`|`) or use any form of delimiter to delimit the values, concatenate them, and store them as a delimited string. There's no limitation on the number of strings stored in an array. Storing complex values as strings bypasses the complex collection limitation.

To illustrate, assume you have a `"searchScope"` array with more than 3,000 elements:

JSON

```
"searchScope": [  
  {  
    "countryCode": "FRA",  
    "productCode": 1234,  
    "categoryCode": "C100"  
  },  
  {  
    "countryCode": "USA",  
    "productCode": 1235,  
    "categoryCode": "C200"  
  }  
  . . .  
]
```

The workaround for storing the values as a delimited string might look like this:

JSON

```
"searchScope": [
    "|FRA|1234|C100|",
    "|FRA|*|*|",
    "|*|1234|*|",
    "|*|*|C100|",
    "|FRA|*|C100|",
    "|*|1234|C100|"
]
```

Storing all of the search variants in the delimited string is helpful in search scenarios where you want to search for items that have just "FRA" or "1234" or another combination within the array.

Here's a filter formatting snippet in C# that converts inputs into searchable strings:

C#

```
foreach (var filterItem in filterCombinations)
{
    var formattedCondition = $"searchScope/any(s: s eq '{filterItem}')";
    combFilter.Append(combFilter.Length > 0 ? " or (" + formattedCondition
+ ")" : "(" + formattedCondition + ")");
}
```

The following list provides inputs and search strings (outputs) side by side:

- For "FRA" county code and the "1234" product code, the formatted output is `|FRA|1234|*|`.
- For "1234" product code, the formatted output is `|*|1234|*|`.
- For "C100" category code, the formatted output is `|*|*|C100|`.

Only provide the wildcard (*) if you're implementing the string array workaround. Otherwise, if you're using a complex type, your filter might look like this example:

C#

```
var countryFilter = $"searchScope/any(ss: search.in(countryCode , 'FRA'))";
var catgFilter = $"searchScope/any(ss: search.in(categoryCode , 'C100'))";
var combinedCountryCategoryFilter = "(" + countryFilter + " and " + catgFilter +
")";
```

If you implement the workaround, be sure to test extensively.

Next steps

Try the [Hotels data set](#) in the **Import data** wizard. You need the Azure Cosmos DB connection information provided in the readme to access the data.

With that information in hand, your first step in the wizard is to create a new Azure Cosmos DB data source. Further on in the wizard, when you get to the target index page, you see an index with complex types. Create and load this index, and then execute queries to understand the new structure.

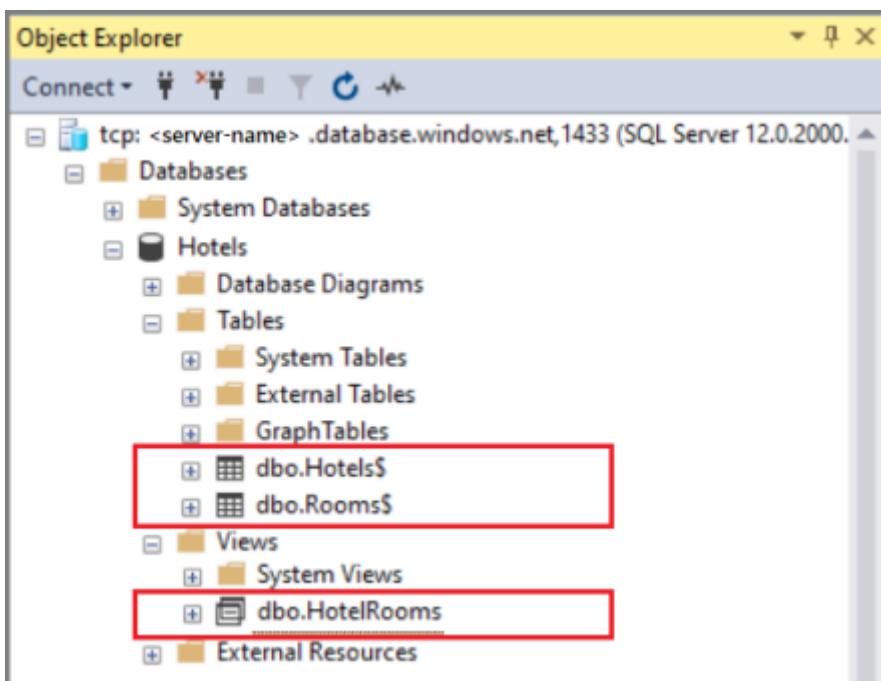
[Quickstart: portal wizard for import, indexing, and queries](#)

How to model relational SQL data for import and indexing in Azure AI Search

07/25/2025

Azure AI Search accepts a flat rowset as input to the [indexing pipeline](#). If your source data originates from joined tables in a SQL Server relational database, this article explains how to construct the rowset, and how to model a parent-child relationship in an Azure AI Search index.

As an illustration, we refer to a hypothetical hotels database, based on [demo data](#). Assume the database consists of a `Hotels$` table with 50 hotels, and a `Rooms$` table with rooms of varying types, rates, and amenities, for a total of 750 rooms. There's a one-to-many relationship between the tables. In our approach, a view provides the query that returns 50 rows, one row per hotel, with associated room detail embedded into each row.



The problem of denormalized data

One of the challenges in working with one-to-many relationships is that standard queries built on joined tables return denormalized data, which doesn't work well in an Azure AI Search scenario. Consider the following example that joins hotels and rooms.

SQL

```
SELECT * FROM Hotels$  
INNER JOIN Rooms$  
ON Rooms$.HotelID = Hotels$.HotelID
```

Results from this query return all of the Hotel fields, followed by all Room fields, with preliminary hotel information repeating for each room value.

Fields from Hotels\$				Fields from Rooms\$				
HotelID	HotelName	Description	State	HotelID	Description	Type	BaseRate	
181	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (City View)	Standard Room	121.99
182	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 1 King Bed (Waterfront View)	Budget Room	88.99
183	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 2 Double Beds (Ctryside)	Standard Room	127.99
184	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 2 Double Beds (Ctryside)	Budget Room	96.99
185	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 1 Queen Bed (Mountain View)	Budget Room	63.99
186	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (Mountain View)	Standard Room	124.99
187	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (Ctryside)	Standard Room	117.99

While this query succeeds on the surface (providing all of the data in a flat rowset), it fails in delivering the right document structure for the expected search experience. During indexing, Azure AI Search creates one search document for each row ingested. If your search documents looked like the above results, you would have perceived duplicates - seven separate documents for the Old Century Hotel alone. A query on "hotels in Florida" would return seven results for just the Old Century Hotel, pushing other relevant hotels deep into the search results.

To get the expected experience of one document per hotel, you should provide a rowset at the right granularity, but with complete information. This article explains how.

Define a query that returns embedded JSON

To deliver the expected search experience, your data set should consist of one row for each search document in Azure AI Search. In our example, we want one row for each hotel, but we also want our users to be able to search on other room-related fields they care about, such as the nightly rate, size and number of beds, or a view of the beach, all of which are part of a room detail.

The solution is to capture the room detail as nested JSON, and then insert the JSON structure into a field in a view, as shown in the second step.

1. Assume you have two joined tables, `Hotels$` and `Rooms$`, that contain details for 50 hotels and 750 rooms and are joined on the `HotelID` field. Individually, these tables contain 50 hotels and 750 related rooms.

SQL

```
CREATE TABLE [dbo].[Hotels$](
    [HotelID] [nchar](10) NOT NULL,
    [HotelName] [nvarchar](255) NULL,
    [Description] [nvarchar](max) NULL,
    [Description_fr] [nvarchar](max) NULL,
    [Category] [nvarchar](255) NULL,
    [Tags] [nvarchar](255) NULL,
    [ParkingIncluded] [float] NULL,
```

```

[SmokingAllowed] [float] NULL,
[LastRenovationDate] [smalldatetime] NULL,
[Rating] [float] NULL,
[StreetAddress] [nvarchar](255) NULL,
[City] [nvarchar](255) NULL,
[State] [nvarchar](255) NULL,
[ZipCode] [nvarchar](255) NULL,
[GeoCoordinates] [nvarchar](255) NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

CREATE TABLE [dbo].[Rooms$](
[HotelID] [nchar](10) NULL,
[Description] [nvarchar](255) NULL,
[Description_fr] [nvarchar](255) NULL,
[Type] [nvarchar](255) NULL,
[BaseRate] [float] NULL,
[BedOptions] [nvarchar](255) NULL,
[SleepsCount] [float] NULL,
[SmokingAllowed] [float] NULL,
[Tags] [nvarchar](255) NULL
) ON [PRIMARY]
GO

```

2. Create a view composed of all fields in the parent table (`SELECT * from dbo.Hotels$`), with the addition of a new *Rooms* field that contains the output of a nested query. A **FOR JSON AUTO** clause on `SELECT * from dbo.Rooms$` structures the output as JSON.

SQL

```

CREATE VIEW [dbo].[HotelRooms]
AS
SELECT *, (SELECT *
            FROM dbo.Rooms$
           WHERE dbo.Rooms$.HotelID = dbo.Hotels$.HotelID FOR JSON AUTO) AS
Rooms
FROM dbo.Hotels$
GO

```

The following screenshot shows the resulting view, with the *Rooms* nvarchar field at the bottom. The *Rooms* field exists only in the HotelRooms view.

The screenshot shows the SQL Server Object Explorer with the following hierarchy:

- Views
- + System Views
- + dbo.HotelRooms
- + Columns
- + HotelID (nchar(10), not null)
- + HotelName (nvarchar(255), null)
- + Description (nvarchar(max), null)
- + Description_fr (nvarchar(max), null)
- + Category (nvarchar(255), null)
- + Tags (nvarchar(255), null)
- + ParkingIncluded (float, null)
- + SmokingAllowed (float, null)
- + LastRenovationDate (smalldatetime, null)
- + Rating (float, null)
- + StreetAddress (nvarchar(255), null)
- + City (nvarchar(255), null)
- + State (nvarchar(255), null)
- + ZipCode (nvarchar(255), null)
- + GeoCoordinates (nvarchar(255), null)
- + Rooms (nvarchar(max), null) Rooms

3. Run `SELECT * FROM dbo.HotelRooms` to retrieve the row set. This query returns 50 rows, one per hotel, with associated room information as a JSON collection.

The screenshot shows the SQL Server Management Studio Results pane with the following data:

HotelID	HotelName	Des...	De...	Ca...	T...	P...	S...	L...	R...	St...	C...	St...	Zi...	Geo...	Rooms
1	Secret Point Motel	Th...	L'	B...	p...	0	1	1...	3...	6...	N.	NY	1...	[7...	[{"HotelID": "1", "Description": "Budget Room, 1 Queen Bed (Cityside)", "De...
2	10 Countryside Hotel	Sa...	É...	B...	2...	0	1	1...	2...	6...	D.	NC	2...	[7...	[{"HotelID": "10", "Description": "Suite, 1 King Bed (Amenities)", "Descri...
3	11 Regal Orb Resort & Spa	Yo...	V...	E...	fr...	1	0	1...	2...	2...	B.	W...	9...	[1...	[{"HotelID": "11", "Description": "Deluxe Room, 1 Queen Bed (Waterfront Vie...
4	12 Winter Panorama Resort	Ne...	R...	I...	l...	0	0	1...	4...	9...	W	OR	9...	[1...	[{"HotelID": "12", "Description": "Deluxe Room, 1 King Bed (Cityside)", "Descri...
5	13 Historic Lion Resort	Un...	U...	B...	v...	0	1	1...	4...	3...	S.	M...	6...	[9...	[{"HotelID": "13", "Description": "Standard Room, 1 King Bed (Mountain Vie...
6	14 Twin Vertex Hotel	Ne...	N...	E...	b...	0	0	1...	4...	1...	D.	TX	7...	[9...	[{"HotelID": "14", "Description": "Budget Room, 1 King Bed (Cityside)", "Desc...
7	15 Peaceful Market Hotel & Spa	Bo...	R...	R...	c...	1	0	2...	3...	1...	N.	NY	1...	[7...	[{"HotelID": "15", "Description": "Standard Room, 1 King Bed (Waterfront Vie...
8	16 Double Sanctuary Resort	5*	5...	R...	v...	0	0	1...	4...	2...	S.	W...	9...	[1...	[{"HotelID": "16", "Description": "Suite, 2 Queen Beds (Amenities)", "Descri...
9	17 Antiquity Hotel	Ele...	É...	B...	r...	0	0	1...	4...	8...	N.	NY	1...	[7...	[{"HotelID": "17", "Description": "Budget Room, 2 Queen Beds (Waterfront Vi...
10	18 Oceanside Resort	Ne...	N...	B...	v...	1	1	1...	4...	5...	T.	FL	3...	[8...	[{"HotelID": "18", "Description": "Standard Room, 1 Queen Bed (Cityside)", ...
11	19 Universe Motel	Bo...	R...	S...	r...	0	0	1...	2...	1...	R.	W...	9...	[1...	[{"HotelID": "19", "Description": "Deluxe Room, 1 Queen Bed (Waterfront Vie...", ...
12	2 Twin Dome Motel	Th...	L'	B...	p...	0	1	1...	3...	1...	S.	FL	3...	[8...	[{"HotelID": "2", "Description": "Suite, 2 Double Beds (Mountain View)", "De...

Query executed successfully. tcp:azs-playground.database... findable (111) | Hotels | 00:00:00 | 50 rows

This rowset is now ready for import into Azure AI Search.

! Note

This approach assumes that embedded JSON is under the [maximum column size limits of SQL Server](#).

Use a complex collection for the "many" side of a one-to-many relationship

On the Azure AI Search side, create an index schema that models the one-to-many relationship using nested JSON. The result set you created in the previous section generally corresponds to the index schema provided next (we cut some fields for brevity).

The following example is similar to the example in [How to model complex data types](#). The *Rooms* structure, which has been the focus of this article, is in the fields collection of an index named *hotels*. This example also shows a complex type for *Address*, which differs from *Rooms* in that it's composed of a fixed set of items, as opposed to the multiple, arbitrary number of items allowed in a collection.

JSON

```
{  
    "name": "hotels",  
    "fields": [  
        { "name": "HotelId", "type": "Edm.String", "key": true, "filterable": true },  
        { "name": "HotelName", "type": "Edm.String", "searchable": true, "filterable":  
false },  
        { "name": "Description", "type": "Edm.String", "searchable": true, "analyzer":  
"en.lucene" },  
        { "name": "Description_fr", "type": "Edm.String", "searchable": true,  
"analyzer": "fr.lucene" },  
        { "name": "Category", "type": "Edm.String", "searchable": true, "filterable":  
true, "facetable": true },  
        { "name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true,  
"facetable": true },  
        { "name": "Tags", "type": "Collection(Edm.String)", "searchable": true,  
"filterable": true, "facetable": true },  
        { "name": "Address", "type": "Edm.ComplexType",  
            "fields": [  
                { "name": "StreetAddress", "type": "Edm.String", "filterable": false,  
"sortable": false, "facetable": false, "searchable": true },  
                { "name": "City", "type": "Edm.String", "searchable": true, "filterable":  
true, "sortable": true, "facetable": true },  
                { "name": "StateProvince", "type": "Edm.String", "searchable": true,  
"filterable": true, "sortable": true, "facetable": true }  
            ]  
        },  
        { "name": "Rooms", "type": "Collection(Edm.ComplexType)",  
            "fields": [  
                { "name": "Description", "type": "Edm.String", "searchable": true,  
"analyzer": "en.lucene" },  
                { "name": "Description_fr", "type": "Edm.String", "searchable": true,  
"analyzer": "fr.lucene" },  
                { "name": "Type", "type": "Edm.String", "searchable": true },  
                { "name": "BaseRate", "type": "Edm.Double", "filterable": true,  
"facetable": true },  
                { "name": "BedOptions", "type": "Edm.String", "searchable": true,  
"filterable": true, "facetable": false },  
                { "name": "SleepsCount", "type": "Edm.Int32", "filterable": true,  
"facetable": true },  
                { "name": "SmokingAllowed", "type": "Edm.Boolean", "filterable": true,  
"facetable": false},  
                { "name": "Tags", "type": "Edm.Collection", "searchable": true }  
            ]  
        }  
    ]  
}
```

```
]  
}
```

Given the previous result set and the above index schema, you have all the required components for a successful indexing operation. The flattened data set meets indexing requirements yet preserves detail information. In the Azure AI Search index, search results fall easily into hotel-based entities, while preserving the context of individual rooms and their attributes.

Facet behavior on complex type subfields

Fields that have a parent, such as the fields under Address and Rooms, are called *subfields*. Although you can assign a "facetable" attribute to a subfield, the count of the facet is always for the main document.

For complex types like Address, where there's just one "Address/City" or "Address/stateProvince" in the document, the facet behavior works as expected. However, in the case of Rooms, where there are multiple subdocuments for each main document, the facet counts can be misleading.

As noted in [Model complex types](#): "the document counts returned in the facet results are calculated for the parent document (a hotel), not the subdocuments in a complex collection (rooms). For example, suppose a hotel has 20 rooms of type "suite". Given this facet parameter facet=Rooms/Type, the facet count is one for the hotel, not 20 for the rooms."

Next steps

Using your own data set, you can use the [Import data wizard](#) to create and load the index. The wizard detects the embedded JSON collection, such as the one contained in *Rooms*, and infers an index schema that includes a complex type collection.

Dashboard > Import data

Import data

* Connect to your data Add cognitive search (Optional) * Customize target index

* Index name ⓘ azuresql-hotels-index ✓

* Key ⓘ HotelID

+ Add field + Add subfield ⌂ Delete

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACTETABLE	SEARCHABLE
HotelID	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
ZipCode	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
GeoCoordinates	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

▼ Rooms Collection(... ^

HotellID	Edm.ComplexType	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Collection(Edm.ComplexType)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
		Collection(Edm.ComplexType)			

Try the following quickstart to learn the basic steps of the Import data wizard.

[Quickstart: Create a search index using Azure portal](#)

Analyzers for text processing in Azure AI Search

07/11/2025

An *analyzer* is a component of the [full text search engine](#) that's responsible for processing strings during indexing and query execution. Text processing (also known as lexical analysis) is transformative, modifying a string through actions such as these:

- Remove non-essential words ([stopwords](#)) and punctuation
- Split up phrases and hyphenated words into component parts
- Lower-case any upper-case words
- Reduce words into primitive root forms for storage efficiency and so that matches can be found regardless of tense

The output of a lexical analyzer is a sequence of [tokens](#).

Lexical analysis applies to `Edm.String` fields that are marked as "searchable", which indicates full text search. For fields of this configuration, analysis occurs during indexing when tokens are created, and then again during query execution when queries are parsed and the engine scans for matching tokens. A match is more likely to occur when the same analyzer is used for both indexing and queries, but you can set the analyzer for each workload independently, depending on your requirements.

Query types that are *not* full text search, such as filters or fuzzy search, don't go through the analysis phase on the query side. Instead, the parser sends those strings directly to the search engine, using the pattern that you provide as the basis for the match. Typically, these query forms require whole-string tokens to make pattern matching work. To ensure whole term tokens are preserved during indexing, you might need [custom analyzers](#). For more information about when and why query terms are analyzed, see [Full text search in Azure AI Search](#).

For more background on lexical analysis, listen to the following video clip for a brief explanation.

https://www.youtube-nocookie.com/embed/Y_X6USgvB1g?version=3&start=132&end=189

Default analyzer

In Azure AI Search, an analyzer is automatically invoked on all string fields marked as searchable.

By default, Azure AI Search uses the [Apache Lucene Standard analyzer \(standard lucene\)](#), which breaks text into elements following the "Unicode Text Segmentation" rules. The standard analyzer converts all characters to their lower case form. Both indexed documents and search terms go through the analysis during indexing and query processing.

You can override the default on a field-by-field basis. Alternative analyzers are:

- [language analyzer](#) for linguistic processing
- [custom analyzer](#) for custom configurations
- [built-in analyzers](#) for as-is usage

Types of analyzers

The following list describes which analyzers are available in Azure AI Search.

[] [Expand table](#)

Category	Description
Standard Lucene analyzer	Default. No specification or configuration is required. This general-purpose analyzer performs well for many languages and scenarios.
Built-in analyzers	Consumed as-is and referenced by name. There are two types: language and language-agnostic. Specialized (language-agnostic) analyzers are used when text inputs require specialized processing or minimal processing. Examples of analyzers in this category include Asciifolding , Keyword , Pattern , Simple , Stop , Whitespace . Language analyzers are used when you need rich linguistic support for individual languages. Azure AI Search supports 35 Lucene language analyzers and 50 Microsoft natural language processing analyzers.
Custom analyzers	Refers to a user-defined configuration of a combination of existing elements, consisting of one tokenizer (required) and optional filters (char or token).

A few built-in analyzers, such as [Pattern](#) or [Stop](#), support a limited set of configuration options. To set these options, create a custom analyzer, consisting of the built-in analyzer and one of the alternative options documented in [Built-in analyzers](#). As with any custom configuration, provide your new configuration with a name, such as `myPatternAnalyzer` to distinguish it from the Lucene Pattern analyzer.

Specifying analyzers

Setting an analyzer is optional. As a general rule, try using the default standard Lucene analyzer first to see how it performs. If queries fail to return the expected results, switching to a different analyzer is often the right solution.

1. If you're using a custom analyzer, add it to the search index under the "analyzer" section.

For more information, see [Create Index](#) and also [Add custom analyzers](#).

2. When defining a field, set its "analyzer" property to one of the following: a [built-in analyzer](#) such as `keyword`, a [language analyzer](#) such as `en.microsoft`, or a custom analyzer (defined in the same index schema).

JSON

```
"fields": [  
  {  
    "name": "Description",  
    "type": "Edm.String",  
    "retrievable": true,  
    "searchable": true,  
    "analyzer": "en.microsoft",  
    "indexAnalyzer": null,  
    "searchAnalyzer": null  
  },
```

3. If you're using a [language analyzer](#), you must use the "analyzer" property to specify it. The "searchAnalyzer" and "indexAnalyzer" properties don't apply to language analyzers.

4. Alternatively, set "indexAnalyzer" and "searchAnalyzer" to vary the analyzer for each workload. These properties work together as a substitute for the "analyzer" property, which must be null. You might use different analyzers for indexing and queries if one of those activities required a specific transformation not needed by the other.

JSON

```
"fields": [  
  {  
    "name": "ProductGroup",  
    "type": "Edm.String",  
    "retrievable": true,  
    "searchable": true,  
    "analyzer": null,  
    "indexAnalyzer": "keyword",  
    "searchAnalyzer": "standard"  
  },
```

When to add analyzers

The best time to add and assign analyzers is during active development, when dropping and recreating indexes is routine.

Because analyzers are used to tokenize terms, you should assign an analyzer when the field is created. In fact, assigning an analyzer or indexAnalyzer to a field that has already been physically created isn't allowed (although you can change the searchAnalyzer property at any time with no impact to the index).

To change the analyzer of an existing field, you'll have to drop and recreate the entire index (you can't rebuild individual fields). For indexes in production, you can defer a rebuild by creating a new field with the new analyzer assignment, and start using it in place of the old one. Use [Update Index](#) to incorporate the new field and [mergeOrUpload](#) to populate it. Later, as part of planned index servicing, you can clean up the index to remove obsolete fields.

To add a new field to an existing index, call [Update Index](#) to add the field, and [mergeOrUpload](#) to populate it.

To add a custom analyzer to an existing index, pass the "allowIndexDowntime" flag in [Update Index](#) if you want to avoid this error:

```
"Index update not allowed because it would cause downtime. In order to add new analyzers, tokenizers, token filters, or character filters to an existing index, set the 'allowIndexDowntime' query parameter to 'true' in the index update request. Note that this operation will put your index offline for at least a few seconds, causing your indexing and query requests to fail. Performance and write availability of the index can be impaired for several minutes after the index is updated, or longer for very large indexes."
```

Recommendations for working with analyzers

This section offers advice on how to work with analyzers.

One analyzer for read-write unless you have specific requirements

Azure AI Search lets you specify different analyzers for indexing and search through the "indexAnalyzer" and "searchAnalyzer" field properties. If unspecified, the analyzer set with the analyzer property is used for both indexing and searching. If the analyzer is unspecified, the default Standard Lucene analyzer is used.

A general rule is to use the same analyzer for both indexing and querying, unless specific requirements dictate otherwise. Be sure to test thoroughly. When text processing differs at

search and indexing time, you run the risk of mismatch between query terms and indexed terms when the search and indexing analyzer configurations aren't aligned.

Test during active development

Overriding the standard analyzer requires an index rebuild. If possible, decide on which analyzers to use during active development, before rolling an index into production.

Inspect tokenized terms

If a search fails to return expected results, the most likely scenario is token discrepancies between term inputs on the query, and tokenized terms in the index. If the tokens aren't the same, matches fail to materialize. To inspect tokenizer output, we recommend using the [Analyze API](#) as an investigation tool. The response consists of tokens, as generated by a specific analyzer.

REST examples

The examples below show analyzer definitions for a few key scenarios.

- [Custom analyzer example](#)
- [Assign analyzers to a field example](#)
- [Mixing analyzers for indexing and search](#)
- [Language analyzer example](#)

Custom analyzer example

This example illustrates an analyzer definition with custom options. Custom options for char filters, tokenizers, and token filters are specified separately as named constructs, and then referenced in the analyzer definition. Predefined elements are used as-is and referenced by name.

Walking through this example:

- Analyzers are a property of the field class for a searchable field.
- A custom analyzer is part of an index definition. It might be lightly customized (for example, customizing a single option in one filter) or customized in multiple places.
- In this case, the custom analyzer is "my_analyzer", which in turn uses a customized standard tokenizer "my_standard_tokenizer" and two token filters: lowercase and customized asciifolding filter "my_asciifolding".

- It also defines 2 custom char filters "map_dash" and "remove_whitespace". The first one replaces all dashes with underscores while the second one removes all spaces. Spaces need to be UTF-8 encoded in the mapping rules. The char filters are applied before tokenization and will affect the resulting tokens (the standard tokenizer breaks on dash and spaces but not on underscore).

JSON

```
{
  "name": "myindex",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": false
    },
    {
      "name": "text",
      "type": "Edm.String",
      "searchable": true,
      "analyzer": "my_analyzer"
    }
  ],
  "analyzers": [
    {
      "name": "my_analyzer",
      "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
      "charFilters": [
        "map_dash",
        "remove_whitespace"
      ],
      "tokenizer": "my_standard_tokenizer",
      "tokenFilters": [
        "my_asciifolding",
        "lowercase"
      ]
    }
  ],
  "charFilters": [
    {
      "name": "map_dash",
      "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
      "mappings": [ "-=>_"]
    },
    {
      "name": "remove_whitespace",
      "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
      "mappings": [ "\u0020=>"]
    }
  ],
  "tokenizers": [
  ]}
```

```
        "name": "my_standard_tokenizer",
        "@odata.type": "#Microsoft.Azure.Search.StandardTokenizerV2",
        "maxTokenLength": 20
    },
],
"tokenFilters": [
{
    "name": "my_asciifolding",
    "@odata.type": "#Microsoft.Azure.Search.AsciiFoldingTokenFilter",
    "preserveOriginal": true
}
]
}
```

Per-field analyzer assignment example

The Standard analyzer is the default. Suppose you want to replace the default with a different predefined analyzer, such as the pattern analyzer. If you aren't setting custom options, you only need to specify it by name in the field definition.

The "analyzer" element overrides the Standard analyzer on a field-by-field basis. There is no global override. In this example, `text1` uses the pattern analyzer and `text2`, which doesn't specify an analyzer, uses the default.

JSON

```
{
  "name": "myindex",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": false
    },
    {
      "name": "text1",
      "type": "Edm.String",
      "searchable": true,
      "analyzer": "pattern"
    },
    {
      "name": "text2",
      "type": "Edm.String",
      "searchable": true
    }
  ]
}
```

Mixing analyzers for indexing and search operations

The APIs include index attributes for specifying different analyzers for indexing and search. The searchAnalyzer and indexAnalyzer attributes must be specified as a pair, replacing the single analyzer attribute.

JSON

```
{  
  "name": "myindex",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true,  
      "searchable": false  
    },  
    {  
      "name": "text",  
      "type": "Edm.String",  
      "searchable": true,  
      "indexAnalyzer": "whitespace",  
      "searchAnalyzer": "simple"  
    },  
  ],  
}
```

Language analyzer example

Fields containing strings in different languages can use a language analyzer, while other fields retain the default (or use some other predefined or custom analyzer). If you use a language analyzer, it must be used for both indexing and search operations. Fields that use a language analyzer can't have different analyzers for indexing and search.

JSON

```
{  
  "name": "myindex",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true,  
      "searchable": false  
    },  
    {  
      "name": "text",  
      "type": "Edm.String",  
      "searchable": true,  
      "indexAnalyzer": "language",  
      "searchAnalyzer": "language"  
    },  
  ],  
}
```

```
        "indexAnalyzer": "whitespace",
        "searchAnalyzer": "simple"
    },
{
    "name": "text_fr",
    "type": "Edm.String",
    "searchable": true,
    "analyzer": "fr.lucene"
}
],
}
```

C# examples

If you are using the .NET SDK code samples, you can append these examples to use or configure analyzers.

- [Assign a built-in analyzer](#)
- [Configure an analyzer](#)

Assign a language analyzer

Any analyzer that is used as-is, with no configuration, is specified on a field definition. There is no requirement for creating an entry in the [analyzers] section of the index.

Language analyzers are used as-is. To use them, call [LexicalAnalyzer](#), specifying the [LexicalAnalyzerName](#) type providing a text analyzer supported in Azure AI Search.

Custom analyzers are similarly specified on the field definition, but for this to work you must specify the analyzer in the index definition, as described in the next section.

C#

```
public partial class Hotel
{
    ...
    [SearchableField(AnalyzerName = LexicalAnalyzerName.EnLucene)]
    public string Description { get; set; }

    [SearchableField(AnalyzerName = LexicalAnalyzerName.FrLucene)]
    [JsonPropertyName("Description_fr")]
    public string DescriptionFr { get; set; }

    [SearchableField(AnalyzerName = "url-analyze")]
    public string Url { get; set; }
    ...
}
```

Define a custom analyzer

When customization or configuration is required, add an analyzer construct to an index. Once you define it, you can add it to the field definition as demonstrated in the previous example.

Create a [CustomAnalyzer](#) object. A custom analyzer is a user-defined combination of a known tokenizer, zero or more token filter, and zero or more character filter names:

- [CustomAnalyzer.Tokenizer](#)
- [CustomAnalyzer.TokenFilters](#)
- [CustomAnalyzer.CharFilters](#)

The following example creates a custom analyzer named "url-analyze" that uses the [uax_url_email](#) tokenizer and the [Lowercase](#) token filter.

C#

```
private static void CreateIndex(string indexName, SearchIndexClient adminClient)
{
    FieldBuilder fieldBuilder = new FieldBuilder();
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var analyzer = new CustomAnalyzer("url-analyze",
LexicalTokenizerName.UaxUrlEmail)
    {
        TokenFilters = { LexicalTokenFilterName.Lowercase }
    };

    var definition = new SearchIndex(indexName, searchFields);

    definition.Analyzers.Add(analyzer);

    adminClient.CreateOrUpdateIndex(definition);
}
```

Next steps

A detailed description of query execution can be found in [Full text search in Azure AI Search](#). The article uses examples to explain behaviors that might seem counter-intuitive on the surface.

To learn more about analyzers, see the following articles:

- [Add a language analyzer](#)
- [Add a custom analyzer](#)
- [Create a search index](#)

- Create a multi-language index

Add language analyzers to string fields in an Azure AI Search index

06/16/2025

A *language analyzer* is a specific type of [text analyzer](#) that performs lexical analysis using the linguistic rules of the target language. Every searchable string field has an **analyzer** property. If your content consists of translated strings, such as separate fields for English and Chinese text, you could specify language analyzers on each field to access the rich linguistic capabilities of those analyzers.

When to use a language analyzer

You should consider a language analyzer in classic search workflows that don't include large language models and their awareness of linguistic rules and multilingual content.

In class search, you might add a language analyzer when awareness of word or sentence structure adds value to text parsing. A common example is the association of irregular verb forms ("bring" and "brought) or plural nouns ("mice" and "mouse"). Without linguistic awareness, these strings are parsed on physical characteristics alone, which fails to catch the connection. Since large chunks of text are more likely to have this content, fields consisting of descriptions, reviews, or summaries are good candidates for a language analyzer.

You might also consider language analyzers when content consists of non-Western language strings. While the [default analyzer \(Standard Lucene\)](#) is language-agnostic, the concept of using spaces and special characters (hyphens and slashes) to separate strings is more applicable to Western languages than non-Western ones.

For example, in Chinese, Japanese, Korean (CJK), and other Asian languages, a space isn't necessarily a word delimiter. Consider the following Japanese string. Because it has no spaces, a language-agnostic analyzer would likely analyze the entire string as one token, when in fact the string is actually a phrase.

これは私たちの銀河系の中ではもっとも重く明るいクラスの球状星団です。
(This is the heaviest and brightest group of spherical stars in our galaxy.)

For the example above, a successful query would have to include the full token, or a partial token using a suffix wildcard, resulting in an unnatural and limiting search experience.

A better experience is to search for individual words: 明るい (Bright), 私たちの (Our), 銀河系 (Galaxy). Using one of the Japanese analyzers available in Azure AI Search is more likely to unlock this behavior because those analyzers are better equipped at splitting the chunk of text into meaningful words in the target language.

Comparing Lucene and Microsoft Analyzers

Azure AI Search supports 35 language analyzers backed by Lucene, and 50 language analyzers backed by proprietary Microsoft natural language processing technology used in Office and Bing.

Some developers might prefer the open-source solution of Lucene. Lucene language analyzers are faster, but the Microsoft analyzers have advanced capabilities, such as lemmatization, word decompounding (in languages like German, Danish, Dutch, Swedish, Norwegian, Estonian, Finnish, Hungarian, Slovak) and entity recognition (URLs, emails, dates, numbers). If possible, you should run comparisons of both the Microsoft and Lucene analyzers to decide which one is a better fit. You can use [Analyze API](#) to see the tokens generated from a given text using a specific analyzer.

Indexing with Microsoft analyzers is on average two to three times slower than their Lucene equivalents, depending on the language. Search performance shouldn't be significantly affected for average size queries.

English analyzers

The default analyzer is Standard Lucene, which works well for English, but perhaps not as well as Lucene's English analyzer or Microsoft's English analyzer.

- Lucene's English analyzer extends the Standard analyzer. It removes possessives (trailing 's) from words, applies stemming as per Porter Stemming algorithm, and removes English stop words.
- Microsoft's English analyzer performs lemmatization instead of stemming. This means it can handle inflected and irregular word forms much better which results in more relevant search results.

How to specify a language analyzer

Set the analyzer during index creation before it's loaded with data.

1. In the field definition, make sure the field is attributed as "searchable" and is of type Edm.String.

- Set the "analyzer" property to one of the language analyzers from the [supported analyzers list](#).

The "analyzer" property is the only property that will accept a language analyzer, and it's used for both indexing and queries. Other analyzer-related properties ("searchAnalyzer" and "indexAnalyzer") won't accept a language analyzer.

Language analyzers can't be customized. If an analyzer isn't meeting your requirements, create a [custom analyzer](#) with the `microsoft_language_tokenizer` or `microsoft_language_stemming_tokenizer`, and then add filters for pre- and post-tokenization processing.

The following example illustrates a language analyzer specification in an index:

JSON

```
{  
  "name": "hotels-sample-index",  
  "fields": [  
    {  
      "name": "Description",  
      "type": "Edm.String",  
      "retrievable": true,  
      "searchable": true,  
      "analyzer": "en.microsoft",  
      "indexAnalyzer": null,  
      "searchAnalyzer": null  
    },  
    {  
      "name": "Description_fr",  
      "type": "Edm.String",  
      "retrievable": true,  
      "searchable": true,  
      "analyzer": "fr.microsoft",  
      "indexAnalyzer": null,  
      "searchAnalyzer": null  
    },  
  ]}
```

For more information about creating an index and setting field properties, see [Create Index \(REST\)](#). For more information about text analysis, see [Analyzers in Azure AI Search](#).

Supported language analyzers

Below is the list of supported languages, with Lucene and Microsoft analyzer names.

[] Expand table

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Arabic	ar.microsoft	ar.lucene
Armenian		hy.lucene
Bangla	bn.microsoft	
Basque		eu.lucene
Bulgarian	bg.microsoft	bg.lucene
Catalan	ca.microsoft	ca.lucene
Chinese Simplified	zh-Hans.microsoft	zh-Hans.lucene
Chinese Traditional	zh-Hant.microsoft	zh-Hant.lucene
Croatian	hr.microsoft	
Czech	cs.microsoft	cs.lucene
Danish	da.microsoft	da.lucene
Dutch	nl.microsoft	nl.lucene
English	en.microsoft	en.lucene
Estonian	et.microsoft	
Finnish	fi.microsoft	fi.lucene
French	fr.microsoft	fr.lucene
Galician		gl.lucene
German	de.microsoft	de.lucene
Greek	el.microsoft	el.lucene
Gujarati	gu.microsoft	
Hebrew	he.microsoft	
Hindi	hi.microsoft	hi.lucene
Hungarian	hu.microsoft	hu.lucene
Icelandic	is.microsoft	
Indonesian (Bahasa)	id.microsoft	id.lucene
Irish		ga.lucene

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Italian	it.microsoft	it.lucene
Japanese	ja.microsoft	ja.lucene
Kannada	kn.microsoft	
Korean	ko.microsoft	ko.lucene
Latvian	lv.microsoft	lv.lucene
Lithuanian	lt.microsoft	
Malayalam	ml.microsoft	
Malay (Latin)	ms.microsoft	
Marathi	mr.microsoft	
Norwegian	nb.microsoft	no.lucene
Persian		fa.lucene
Polish	pl.microsoft	pl.lucene
Portuguese (Brazil)	pt-Br.microsoft	pt-Br.lucene
Portuguese (Portugal)	pt-Pt.microsoft	pt-Pt.lucene
Punjabi	pa.microsoft	
Romanian	ro.microsoft	ro.lucene
Russian	ru.microsoft	ru.lucene
Serbian (Cyrillic)	sr-cyrillic.microsoft	
Serbian (Latin)	sr-latin.microsoft	
Slovak	sk.microsoft	
Slovenian	sl.microsoft	
Spanish	es.microsoft	es.lucene
Swedish	sv.microsoft	sv.lucene
Tamil	ta.microsoft	
Telugu	te.microsoft	
Thai	th.microsoft	th.lucene

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Turkish	tr.microsoft	tr.lucene
Ukrainian	uk.microsoft	
Urdu	ur.microsoft	
Vietnamese	vi.microsoft	

All analyzers with names annotated with **Lucene** are powered by [Apache Lucene's language analyzers](#).

See also

- [Create an index](#)
- [Create a multi-language index](#)
- [Create Index \(REST API\)](#)
- [LexicalAnalyzerName Class \(Azure SDK for .NET\)](#)

Add custom analyzers to string fields in an Azure AI Search index

06/16/2025

A *custom analyzer* is a component of lexical analysis over plain text content. It's a user-defined combination of one tokenizer, one or more token filters, and one or more character filters. A custom analyzer is specified within a search index, and then referenced by name on field definitions that require custom analysis. A custom analyzer is invoked on a per-field basis. Attributes on the field determine whether it's used for indexing, queries, or both.

In a custom analyzer, character filters prepare the input text before it's processed by the tokenizer (for example, removing markup). Next, the tokenizer breaks text into tokens. Finally, token filters modify the tokens emitted by the tokenizer. For concepts and examples, see [Analyzers in Azure AI Search](#) and [Tutorial: Create a custom analyzer for phone numbers](#).

Why use a custom analyzer?

You should consider a custom analyzer in classic search workflows that don't include large language models and their ability to handle content anomalies.

In class search, a custom analyzer gives you control over the process of converting plain text into indexable and searchable tokens by allowing you to choose which types of analysis or filtering to invoke, and the order in which they occur.

Create and assign a custom analyzer if none of default (Standard Lucene), built-in, or language analyzers are sufficient for your needs. You might also create a custom analyzer if you want to use a built-in analyzer with custom options. For example, if you wanted to change the `maxTokenLength` on Standard Lucene, you would create a custom analyzer, with a user-defined name, to set that option.

Scenarios where custom analyzers can be helpful include:

- Using character filters to remove HTML markup before text inputs are tokenized, or replace certain characters or symbols.
- Phonetic search. Add a phonetic filter to enable searching based on how a word sounds, not how it's spelled.
- Disable lexical analysis. Use the Keyword analyzer to create searchable fields that aren't analyzed.

- Fast prefix/suffix search. Add the Edge N-gram token filter to index prefixes of words to enable fast prefix matching. Combine it with the Reverse token filter to do suffix matching.
- Custom tokenization. For example, use the Whitespace tokenizer to break sentences into tokens using whitespace as a delimiter
- ASCII folding. Add the Standard ASCII folding filter to normalize diacritics like ö or ê in search terms.

 **Note**

Custom analyzers aren't exposed in the Azure portal. The only way to add a custom analyzer is through code that [creates an index schema](#).

Create a custom analyzer

To create a custom analyzer, specify it in the `analyzers` section of an index at design time, and then reference it on searchable, `Edm.String` fields using either the `analyzer` property, or the `indexAnalyzer` and `searchAnalyzer` pair.

An analyzer definition includes a name, type, one or more character filters, a maximum of one tokenizer, and one or more token filters for post-tokenization processing. Character filters are applied before tokenization. Token filters and character filters are applied from left to right.

- Names in a custom analyzer must be unique and can't be the same as any of the built-in analyzers, tokenizers, token filters, or characters filters. Names consist of letters, digits, spaces, dashes, or underscores. Names must start and end with plain text characters. Names must be under 128 characters in length.
- Type must be `#Microsoft.Azure.Search.CustomAnalyzer`.
- `charFilters` can be one or more filters from [Character Filters](#), processed before tokenization, in the order provided. Some character filters have options, which can be set in a `charFilters` section. Character filters are optional.
- `tokenizer` is exactly one [Tokenizer](#). A value is required. If you need more than one tokenizer, you can create multiple custom analyzers and assign them on a field-by-field basis in your index schema.
- `tokenFilters` can be one or more filters from [Token Filters](#), processed after tokenization, in the order provided. For token filters that have options, add a `tokenFilter` section to

specify the configuration. Token filters are optional.

Analyzers must not produce tokens longer than 300 characters, or indexing will fail. To trim long token or to exclude them, use the **TruncateTokenFilter** and the **LengthTokenFilter** respectively. See [Token filters](#) for reference.

JSON

```
"analyzers":(optional)[
  {
    "name":"name of analyzer",
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "charFilters":[
      "char_filter_name_1",
      "char_filter_name_2"
    ],
    "tokenizer":"tokenizer_name",
    "tokenFilters":[
      "token_filter_name_1",
      "token_filter_name_2"
    ]
  },
  {
    "name":"name of analyzer",
    "@odata.type": "#analyzer_type",
    "option1":value1,
    "option2":value2,
    ...
  }
],
"charFilters":(optional)[
  {
    "name":"char_filter_name",
    "@odata.type": "#char_filter_type",
    "option1":value1,
    "option2":value2,
    ...
  }
],
"tokenizers":(optional)[
  {
    "name":"tokenizer_name",
    "@odata.type": "#tokenizer_type",
    "option1":value1,
    "option2":value2,
    ...
  }
],
"tokenFilters":(optional)[
  {
    "name":"token_filter_name",
    "@odata.type": "#token_filter_type",
    "option1":value1,
```

```
        "option2":value2,  
        ...  
    }  
]
```

Within an index definition, you can place this section anywhere in the body of a create index request but usually it goes at the end:

JSON

```
{  
    "name": "name_of_index",  
    "fields": [ ],  
    "suggesters": [ ],  
    "scoringProfiles": [ ],  
    "defaultScoringProfile": (optional) "...",  
    "corsOptions": (optional) { },  
    "analyzers":(optional)[ ],  
    "charFilters":(optional)[ ],  
    "tokenizers":(optional)[ ],  
    "tokenFilters":(optional)[ ]  
}
```

The analyzer definition is a part of the larger index. Definitions for char filters, tokenizers, and token filters are added to the index only if you're setting custom options. To use an existing filter or tokenizer as-is, specify it by name in the analyzer definition. For more information, see [Create Index \(REST\)](#). For more examples, see [Add analyzers in Azure AI Search](#).

Test custom analyzers

You can use the [Test Analyzer \(REST\)](#) to see how an analyzer breaks given text into tokens.

Request

HTTP

```
POST https://[search service name].search.windows.net/indexes/[index  
name]/analyze?api-version=[api-version]  
Content-Type: application/json  
api-key: [admin key]  
  
{  
    "analyzer": "my_analyzer",  
    "text": "Vis-à-vis means Opposite"  
}
```

Response

HTTP

```
{  
  "tokens": [  
    {  
      "token": "vis_a_vis",  
      "startOffset": 0,  
      "endOffset": 9,  
      "position": 0  
    },  
    {  
      "token": "vis_à_vis",  
      "startOffset": 0,  
      "endOffset": 9,  
      "position": 0  
    },  
    {  
      "token": "means",  
      "startOffset": 10,  
      "endOffset": 15,  
      "position": 1  
    },  
    {  
      "token": "opposite",  
      "startOffset": 16,  
      "endOffset": 24,  
      "position": 2  
    }  
  ]  
}
```

Update custom analyzers

Once an analyzer, a tokenizer, a token filter, or a character filter is defined, it can't be modified. New ones can be added to an existing index only if the `allowIndexDowntime` flag is set to true in the index update request:

HTTP

```
PUT https://[search service name].search.windows.net/indexes/[index name]?api-version=[api-version]&allowIndexDowntime=true
```

This operation takes your index offline for at least a few seconds, causing your indexing and query requests to fail. Performance and write availability of the index can be impaired for several minutes after the index is updated, or longer for very large indexes, but these effects are temporary and eventually resolve on their own.

Built-in analyzers

If you want to use a built-in analyzer with custom options, creating a custom analyzer is the mechanism by which you specify those options. In contrast, to use a built-in analyzer as-is, you simply need to [reference it by name](#) in the field definition.

[\[+\] Expand table](#)

analyzer_name	analyzer_type ¹	Description and Options
keyword	(type applies only when options are available)	Treats the entire content of a field as a single token. This is useful for data like zip codes, IDs, and some product names.
pattern	PatternAnalyzer	Flexibly separates text into terms via a regular expression pattern. Options lowercase (type: bool) - Determines whether terms are lowercased. The default is true. pattern (type: string) - A regular expression pattern to match token separators. The default is <code>\W+</code> , which matches non-word characters. flags (type: string) - Regular expression flags. The default is an empty string. Allowed values: CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, LITERAL, MULTILINE, UNICODE_CASE, UNIX_LINES stopwords (type: string array) - A list of stopwords. The default is an empty list.
simple	(type applies only when options are available)	Divides text at non-letters and converts them to lower case.
standard (Also referred to as standard.lucene)	StandardAnalyzer	Standard Lucene analyzer, composed of the standard tokenizer, lowercase filter, and stop filter. Options maxTokenLength (type: int) - The maximum token length. The default is 255. Tokens longer than the maximum length are split. Maximum token length that can be used is 300 characters.

analyzer_name	analyzer_type ¹	Description and Options
		stopwords (type: string array) - A list of stopwords. The default is an empty list.
standardascifolding.lucene	(type applies only when options are available)	Standard analyzer with Ascii folding filter.
stop ↗	StopAnalyzer	Divides text at non-letters, applies the lowercase and stopword token filters.
		Options
		stopwords (type: string array) - A list of stopwords. The default is a predefined list for English.
whitespace ↗	(type applies only when options are available)	An analyzer that uses the whitespace tokenizer. Tokens that are longer than 255 characters are split.

¹ Analyzer Types are always prefixed in code with `#Microsoft.Azure.Search` such that `PatternAnalyzer` would actually be specified as `#Microsoft.Azure.Search.PatternAnalyzer`. We removed the prefix for brevity, but the prefix is required in your code.

The analyzer_type is only provided for analyzers that can be customized. If there are no options, as is the case with the keyword analyzer, there's no associated `#Microsoft.Azure.Search` type.

Character filters

Character filters add processing before a string reaches the tokenizer.

Azure AI Search supports character filters in the following list. More information about each one can be found in the Lucene API reference.

 [Expand table](#)

char_filter_name	char_filter_type ¹	Description and Options
html_strip ↗	(type applies only when options are available)	A char filter that attempts to strip out HTML constructs.
mapping ↗	MappingCharFilter	A char filter that applies mappings defined with the mappings option. Matching is greedy (longest pattern

char_filter_name	char_filter_type ¹	Description and Options
		<p>matching at a given point wins). Replacement is allowed to be the empty string.</p> <p>Options</p> <p>mappings (type: string array) - A list of mappings of the following format: <code>a=>b</code> (all occurrences of the character <code>a</code> are replaced with character <code>b</code>). Required.</p>
pattern_replace	PatternReplaceCharFilter	<p>A char filter that replaces characters in the input string. It uses a regular expression to identify character sequences to preserve and a replacement pattern to identify characters to replace. For example, input text = <code>aa bb aa bb</code>, pattern= <code>(aa)\\s+(bb)</code> replacement= <code>\$1#\$2</code>, result = <code>aa#bb aa#bb</code>.</p> <p>Options</p> <p>pattern (type: string) - Required.</p> <p>replacement (type: string) - Required.</p>

¹ Char Filter Types are always prefixed in code with `#Microsoft.Azure.Search` such that `MappingCharFilter` would actually be specified as `#Microsoft.Azure.Search.MappingCharFilter`. We removed the prefix to reduce the width of the table, but remember to include it in your code. Notice that `char_filter_type` is only provided for filters that can be customized. If there are no options, as is the case with `html_strip`, there's no associated `#Microsoft.Azure.Search` type.

Tokenizers

A tokenizer divides continuous text into a sequence of tokens, such as breaking a sentence into words, or a word into root forms.

Azure AI Search supports tokenizers in the following list. More information about each one can be found in the Lucene API reference.

[Expand table](#)

tokenizer_name	tokenizer_type ¹	Description and Options
classic	ClassicTokenizer	Grammar based tokenizer that is suitable for

tokenizer_name	tokenizer_type ¹	Description and Options
		<p>processing most European-language documents.</p> <p>Options</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.</p>
edgeNGram	EdgeNGramTokenizer	<p>Tokenizes the input from an edge into n-grams of given sizes.</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram.</p> <p>tokenChars (type: string array) - Character classes to keep in the tokens. Allowed values: <code>letter</code>, <code>digit</code>, <code>whitespace</code>, <code>punctuation</code>, <code>symbol</code>. Defaults to an empty array - keeps all characters.</p>
keyword_v2	KeywordTokenizerV2	<p>Emits the entire input as a single token.</p> <p>Options</p>

tokenizer_name	tokenizer_type ¹	Description and Options
		maxTokenLength (type: int) - The maximum token length. Default: 256, maximum: 300. Tokens longer than the maximum length are split.
letter	(type applies only when options are available)	Divides text at non-letters. Tokens that are longer than 255 characters are split.
lowercase	(type applies only when options are available)	Divides text at non-letters and converts them to lower case. Tokens that are longer than 255 characters are split.
microsoft_language_tokenizer	MicrosoftLanguageTokenizer	Divides text using language-specific rules.
		<p>Options</p> <p>maxTokenLength (type: int) - The maximum token length, default: 255, maximum: 300. Tokens longer than the maximum length are split. Tokens longer than 300 characters are first split into tokens of length 300 and then each of those tokens is split based on the maxTokenLength set.</p> <p>isSearchTokenizer (type: bool) - Set to true if used as the</p>

tokenizer_name	tokenizer_type ¹	Description and Options
		<p>search tokenizer, set to false if used as the indexing tokenizer.</p> <p>language (type: string) - Language to use, default <code>english</code>. Allowed values include:</p> <pre>bangla, bulgarian, catalan, chineseSimplified, chineseTraditional, croatian, czech, danish, dutch, english, french, german, greek, gujarati, hindi, icelandic, indonesian, italian, japanese, kannada, korean, malay, malayalam, marathi, norwegianBokmaal, polish, portuguese, portugueseBrazilian, punjabi, romanian, russian, serbianCyrillic, serbianLatin, slovenian, spanish, swedish, tamil, telugu, thai, ukrainian, urdu, vietnamese</pre>
microsoft_language_stemming_tokenizer	MicrosoftLanguageStemmingTokenizer	<p>Divides text using language-specific rules and reduces words to their base forms. This tokenizer performs lemmatization.</p> <p>Options</p>

tokenizer_name	tokenizer_type ¹	Description and Options
	<p>maxTokenLength (type: int) - The maximum token length, default: 255, maximum: 300.</p> <p>Tokens longer than the maximum length are split. Tokens longer than 300 characters are first split into tokens of length 300 and then each of those tokens is split based on the maxTokenLength set.</p> <p>isSearchTokenizer (type: bool) - Set to true if used as the search tokenizer, set to false if used as the indexing tokenizer.</p> <p>language (type: string) - Language to use, default <code>english</code>. Allowed values include:</p> <ul style="list-style-type: none"> <code>arabic</code>, <code>bangla</code>, <code>bulgarian</code>, <code>catalan</code>, <code>croatian</code>, <code>czech</code>, <code>danish</code>, <code>dutch</code>, <code>english</code>, <code>estonian</code>, <code>finnish</code>, <code>french</code>, <code>german</code>, <code>greek</code>, <code>gujarati</code>, <code>hebrew</code>, <code>hindi</code>, <code>hungarian</code>, <code>icelandic</code>, <code>indonesian</code>, <code>italian</code>, <code>kannada</code>, <code>latvian</code>, <code>lithuanian</code>, <code>malay</code>, <code>malayalam</code>, <code>marathi</code>, <code>norwegianBokmaal</code>, <code>polish</code>, <code>portuguese</code>, <code>portugueseBrazilian</code>, <code>punjabi</code>, <code>romanian</code>, 	

tokenizer_name	tokenizer_type ¹	Description and Options
		russian, serbianCyrillic, serbianLatin, slovak, slovenian, spanish, swedish, tamil, telugu, turkish, ukrainian, urdu
nGram ↗	NGramTokenizer	<p>Tokenizes the input into n-grams of the given sizes.</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram.</p> <p>tokenChars (type: string array) - Character classes to keep in the tokens. Allowed values: letter, digit, whitespace, punctuation, symbol. Defaults to an empty array - keeps all characters.</p>
path_hierarchy_v2 ↗	PathHierarchyTokenizerV2	<p>Tokenizer for path-like hierarchies.</p> <p>Options</p> <p>delimiter (type: string) - Default: '/.</p> <p>replacement (type: string) - If set, replaces the delimiter character.</p>

tokenizer_name	tokenizer_type ¹	Description and Options
		<p>Default same as the value of delimiter.</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 300, maximum: 300. Paths longer than maxTokenLength are ignored.</p> <p>reverse (type: bool) - If true, generates token in reverse order. Default: false.</p> <p>skip (type: bool) - Initial tokens to skip. The default is 0.</p>
pattern ↴	PatternTokenizer	<p>This tokenizer uses regex pattern matching to construct distinct tokens.</p> <p>Options</p> <p>pattern ↴ (type: string) - Regular expression pattern to match token separators. The default is <code>\W+</code>, which matches non-word characters.</p> <p>flags ↴ (type: string) - Regular expression flags. The default is an empty string. Allowed values: CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, LITERAL,</p>

tokenizer_name	tokenizer_type ¹	Description and Options
	MULTILINE, UNICODE_CASE, UNIX_LINES	
		group (type: int) - Which group to extract into tokens. The default is -1 (split).
standard_v2 ↗	StandardTokenizerV2	Breaks text following the Unicode Text Segmentation rules ↗ .
		Options
		maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.
uax_url_email ↗	UaxUrlEmailTokenizer	Tokenizes urls and emails as one token.
		Options
		maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.
whitespace ↗	(type applies only when options are available)	Divides text at whitespace. Tokens that are longer than 255 characters are split.

¹ Tokenizer Types are always prefixed in code with `#Microsoft.Azure.Search` such that `ClassicTokenizer` would actually be specified as `#Microsoft.Azure.Search.ClassicTokenizer`. We removed the prefix to reduce the width of the table, but remember to include it in your code. Notice that `tokenizer_type` is only provided for tokenizers that can be customized. If there are no options, as is the case with the letter tokenizer, there's no associated `#Microsoft.Azure.Search` type.

Token filters

A token filter is used to filter out or modify the tokens generated by a tokenizer. For example, you can specify a lowercase filter that converts all characters to lowercase. You can have multiple token filters in a custom analyzer. Token filters run in the order in which they're listed.

In the following table, the token filters that are implemented using Apache Lucene are linked to the Lucene API documentation.

[+] Expand table

token_filter_name	token_filter_type ¹	Description and Options
arabic_normalization ↴	(type applies only when options are available)	A token filter that applies the Arabic normalizer to normalize the orthography.
apostrophe ↴	(type applies only when options are available)	Strips all characters after an apostrophe (including the apostrophe itself).
asciifolding ↴	AsciiFoldingTokenFilter	Converts alphabetic, numeric, and symbolic Unicode characters which aren't in the first 127 ASCII characters (the <code>Basic Latin</code> Unicode block) into their ASCII equivalents, if one exists.
		Options
		preserveOriginal (type: bool) - If true, the original token is kept. The default is false.
cjk_bigram ↴	CjkBigramTokenFilter	Forms bigrams of CJK terms that are generated from StandardTokenizer.

<code>token_filter_name</code>	<code>token_filter_type</code> ¹	Description and Options
		<p>Options</p> <p><code>ignoreScripts</code> (type: string array) - Scripts to ignore. Allowed values include: <code>han</code>, <code>hiragana</code>, <code>katakana</code>, <code>hangul</code>. The default is an empty list.</p> <p><code>outputUnigrams</code> (type: bool) - Set to true if you always want to output both unigrams and bigrams. The default is false.</p>
cjk_width	(type applies only when options are available)	Normalizes CJK width differences. Folds full width ASCII variants into the equivalent basic Latin and half-width Katakana variants into the equivalent kana.
classic	(type applies only when options are available)	Removes the English possessives, and dots from acronyms.
common_grams	CommonGramTokenFilter	<p>Construct bigrams for frequently occurring terms while indexing. Single terms are still indexed too, with bigrams overlaid.</p> <p>Options</p> <p><code>commonWords</code> (type: string array) - The set of common words. The default is an empty list. Required.</p> <p><code>ignoreCase</code> (type: bool) - If true, matching is case insensitive. The default is false.</p> <p><code>queryMode</code> (type: bool) - Generates bigrams then removes common words and single terms followed by a common word. The default is false.</p>

token_filter_name	token_filter_type ¹	Description and Options
dictionary_decompounder	DictionaryDecompounderTokenFilter	<p>Decomposes compound words found in many Germanic languages.</p> <p>Options</p> <p>wordList (type: string array) - The list of words to match against. The default is an empty list. Required.</p> <p>minWordSize (type: int) - Only words longer than this will be processed. The default is 5.</p> <p>minSubwordSize (type: int) - Only subwords longer than this will be outputted. The default is 2.</p> <p>maxSubwordSize (type: int) - Only subwords shorter than this will be outputted. The default is 15.</p> <p>onlyLongestMatch (type: bool) - Add only the longest matching subword to output. The default is false.</p>
edgeNGram_v2	EdgeNGramTokenFilterV2	<p>Generates n-grams of the given sizes from starting from the front or the back of an input token.</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram.</p> <p>side (type: string) - Specifies which side of the input the n-gram should be generated</p>

<code>token_filter_name</code>	<code>token_filter_type</code> ¹	Description and Options
		from. Allowed values: <code>front</code> , <code>back</code>
elision	ElisionTokenFilter	<p>Removes elisions. For example, <code>l'avion</code> (the plane) is converted to <code>avion</code> (plane).</p> <p>Options</p> <p><code>articles</code> (type: string array) - A set of articles to remove. The default is an empty list. If there's no list of articles set, by default all French articles are removed.</p>
german_normalization	(type applies only when options are available)	Normalizes German characters according to the heuristics of the German2 snowball algorithm .
hindi_normalization	(type applies only when options are available)	Normalizes text in Hindi to remove some differences in spelling variations.
indic_normalization	IndicNormalizationTokenFilter	Normalizes the Unicode representation of text in Indian languages.
keep	KeepTokenFilter	<p>A token filter that only keeps tokens with text contained in specified list of words.</p> <p>Options</p> <p><code>keepWords</code> (type: string array) - A list of words to keep. The default is an empty list. Required.</p> <p><code>keepWordsCase</code> (type: bool) - If true, lower case all words first. The default is false.</p>
keyword_marker	KeywordMarkerTokenFilter	Marks terms as keywords.
		Options
		<code>keywords</code> (type: string array) -

token_filter_name	token_filter_type ¹	Description and Options
		A list of words to mark as keywords. The default is an empty list. Required.
		ignoreCase (type: bool) - If true, lower case all words first. The default is false.
keyword_repeat ↗	(type applies only when options are available)	Emits each incoming token twice once as keyword and once as non-keyword.
kstem ↗	(type applies only when options are available)	A high-performance <code>kstem</code> filter for English.
length ↗	LengthTokenFilter	Removes words that are too long or too short.
		Options
		min (type: int) - The minimum number. Default: 0, maximum: 300.
		max (type: int) - The maximum number. Default: 300, maximum: 300.
limit ↗	Microsoft.Azure.Search.LimitTokenFilter	Limits the number of tokens while indexing.
		Options
		maxTokenCount (type: int) - Max number of tokens to produce. The default is 1.
		consumeAllTokens (type: bool) - Whether all tokens from the input must be consumed even if maxTokenCount is reached. The default is false.
lowercase ↗	(type applies only when options are available)	Normalizes token text to lower case.
nGram_v2 ↗	NGramTokenFilterV2	Generates n-grams of the given sizes.

token_filter_name	token_filter_type ¹	Description and Options
		<p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram.</p>
pattern_capture	PatternCaptureTokenFilter	<p>Uses Java regexes to emit multiple tokens, one for each capture group in one or more patterns.</p> <p>Options</p> <p>patterns (type: string array) - A list of patterns to match against each token. Required.</p> <p>preserveOriginal (type: bool) - Set to true to return the original token even if one of the patterns matches, default: true</p>
pattern_replace	PatternReplaceTokenFilter	<p>A token filter which applies a pattern to each token in the stream, replacing match occurrences with the specified replacement string.</p> <p>Options</p> <p>pattern (type: string) - Required.</p> <p>replacement (type: string) - Required.</p>
persian_normalization	(type applies only when options are available)	Applies normalization for Persian.
phonetic	PhoneticTokenFilter	Create tokens for phonetic matches.
		<p>Options</p>

token_filter_name	token_filter_type ¹	Description and Options
		<p>encoder (type: string) - Phonetic encoder to use. Allowed values include: metaphone, doubleMetaphone, soundex, refinedSoundex, caverphone1, caverphone2, cologne, nysiis, koelnerPhonetik, haasePhonetik, beiderMorse. Default: metaphone. Default is metaphone.</p> <p>See encoder for more information.</p>
		<p>replace (type: bool) - True if encoded tokens should replace original tokens, false if they should be added as synonyms. The default is true.</p>
porter_stem	(type applies only when options are available)	Transforms the token stream as per the Porter stemming algorithm .
reverse	(type applies only when options are available)	Reverses the token string.
scandinavian_normalization	(type applies only when options are available)	Normalizes use of the interchangeable Scandinavian characters.
scandinavian_folding	(type applies only when options are available)	Folds Scandinavian characters åÅääää into a and ööøø into o. It also discriminates against use of double vowels aa, ae, ao, oe and oo, leaving just the first one.
shingle	ShingleTokenFilter	<p>Creates combinations of tokens as a single token.</p> <p>Options</p> <p>maxShingleSize (type: int) - Defaults to 2.</p> <p>minShingleSize (type: int) -</p>

<code>token_filter_name</code>	<code>token_filter_type</code> ¹	Description and Options
		Defaults to 2.
		<p><code>outputUnigrams</code> (type: bool) - if true, the output stream contains the input tokens (unigrams) as well as shingles. The default is true.</p> <p><code>outputUnigramsIfNoShingles</code> (type: bool) - If true, override the behavior of <code>outputUnigrams==false</code> for those times when no shingles are available. The default is false.</p> <p><code>tokenSeparator</code> (type: string) - The string to use when joining adjacent tokens to form a shingle. The default is a single empty space <code> </code>.</p> <p><code>filterToken</code> (type: string) - The string to insert for each position for which there's no token. The default is <code>_</code>.</p>
snowball ↗	SnowballTokenFilter	<p>Snowball Token Filter.</p> <p>Options</p> <p><code>language</code> (type: string) - Allowed values include:</p> <ul style="list-style-type: none"> <code>armenian</code>, <code>basque</code>, <code>catalan</code>, <code>danish</code>, <code>dutch</code>, <code>english</code>, <code>finnish</code>, <code>french</code>, <code>german</code>, <code>german2</code>, <code>hungarian</code>, <code>italian</code>, <code>kp</code>, <code>lovins</code>, <code>norwegian</code>, <code>porter</code>, <code>portuguese</code>, <code>romanian</code>, <code>russian</code>, <code>spanish</code>, <code>swedish</code>, <code>turkish</code>
sorani_normalization ↗	SoraniNormalizationTokenFilter	<p>Normalizes the Unicode representation of <code>Sorani</code> text.</p> <p>Options</p> <p>None.</p>

token_filter_name	token_filter_type ¹	Description and Options
stemmer	StemmerTokenFilter	<p>Language-specific stemming filter.</p> <p>Options</p> <p>language (type: string) - Allowed values include:</p> <ul style="list-style-type: none"> - arabic - armenian - basque - brazilian - bulgarian - catalan - czech - danish - dutch - dutchKp - english - lightEnglish - minimalEnglish - possessiveEnglish - porter2 - lovins - finnish - lightFinnish - french - lightFrench - minimalFrench - galician - minimalGalician - german - german2 - lightGerman - minimalGerman - greek - hindi - hungarian - lightHungarian - indonesian - irish - italian - lightItalian - sorani - latvian - norwegian - lightNorwegian - minimalNorwegian

token_filter_name	token_filter_type ¹	Description and Options
		<ul style="list-style-type: none"> - lightNynorsk - minimalNynorsk - portuguese - lightPortuguese - minimalPortuguese - portugueseRslp - romanian - russian - lightRussian - spanish - lightSpanish - swedish - lightSwedish - turkish
stemmer_override	StemmerOverrideTokenFilter	<p>Any dictionary-Stemmed terms are marked as keywords, which prevents stemming down the chain. Must be placed before any stemming filters.</p> <p>Options</p> <p>rules (type: string array) - Stemming rules in the following format <code>word => stem</code> for example <code>ran => run</code>. The default is an empty list. Required.</p>
stopwords	StopwordsTokenFilter	<p>Removes stop words from a token stream. By default, the filter uses a predefined stop word list for English.</p> <p>Options</p> <p>stopwords (type: string array) - A list of stopwords. Can't be specified if a <code>stopwordsList</code> is specified.</p> <p>stopwordsList (type: string) - A predefined list of stopwords. Can't be specified if <code>stopwords</code> is specified. Allowed values include: <code>arabic</code>, <code>armenian</code>, <code>basque</code>, <code>brazilian</code>, <code>bulgarian</code>,</p>

token_filter_name	token_filter_type ¹	Description and Options
		<p>catalan, czech, danish, dutch, english, finnish, french, galician, german, greek, hindi, hungarian, indonesian, irish, italian, latvian, norwegian, persian, portuguese, romanian, russian, sorani, spanish, swedish, thai, turkish, default: english. Can't be specified if stopwords is specified.</p> <p>ignoreCase (type: bool) - If true, all words are lower cased first. The default is false.</p> <p>removeTrailing (type: bool) - If true, ignore the last search term if it's a stop word. The default is true.</p>
synonym ↗	SynonymTokenFilter	<p>Matches single or multi word synonyms in a token stream.</p> <p>Options</p> <p>synonyms (type: string array) - Required. List of synonyms in one of the following two formats:</p> <ul style="list-style-type: none"> -incredible, unbelievable, fabulous => amazing - all terms on the left side of => symbol are replaced with all terms on its right side. -incredible, unbelievable, fabulous, amazing - A comma-separated list of equivalent words. Set the expand option to change how this list is interpreted. <p>ignoreCase (type: bool) - Case-folds input for matching. The default is false.</p>

token_filter_name	token_filter_type ¹	Description and Options
		<p>expand (type: bool) - If true, all words in the list of synonyms (if => notation isn't used) map to one another.</p> <p>The following list: incredible, unbelievable, fabulous, amazing is equivalent to: incredible, unbelievable, fabulous, amazing => incredible, unbelievable, fabulous, amazing</p> <p>- If false, the following list: incredible, unbelievable, fabulous, amazing are equivalent to: incredible, unbelievable, fabulous, amazing => incredible.</p>
trim ↗	(type applies only when options are available)	Trims leading and trailing whitespace from tokens.
truncate ↗	TruncateTokenFilter	<p>Truncates the terms into a specific length.</p> <p>Options</p> <p>length (type: int) - Default: 300, maximum: 300. Required.</p>
unique ↗	UniqueTokenFilter	<p>Filters out tokens with same text as the previous token.</p> <p>Options</p> <p>onlyOnSamePosition (type: bool) - If set, remove duplicates only at the same position. The default is true.</p>
uppercase ↗	(type applies only when options are available)	Normalizes token text to uppercase.
word_delimiter ↗	WordDelimiterTokenFilter	Splits words into subwords and performs optional transformations on subword groups.

token_filter_name	token_filter_type ¹	Description and Options
		Options
		generateWordParts (type: bool) - Causes parts of words to be generated, for example AzureSearch becomes Azure Search. The default is true.
		generateNumberParts (type: bool) - Causes number subwords to be generated. The default is true.
		catenateWords (type: bool) - Causes maximum runs of word parts to be catenated, for example Azure-Search becomes AzureSearch. The default is false.
		catenateNumbers (type: bool) - Causes maximum runs of number parts to be catenated, for example 1-2 becomes 12. The default is false.
		catenateAll (type: bool) - Causes all subword parts to be catenated, e.g. Azure-Search-1 becomes AzureSearch1. The default is false.
		splitOnCaseChange (type: bool) - If true, splits words on caseChange, for example AzureSearch becomes Azure Search. The default is true.
		preserveOriginal - Causes original words to be preserved and added to the subword list. The default is false.
		splitOnNumerics (type: bool) - If true, splits on numbers, for example Azure1Search becomes Azure 1 Search. The

<code>token_filter_name</code>	<code>token_filter_type</code> ¹	Description and Options
		default is true.
		stemEnglishPossessive (type: bool) - Causes trailing 's to be removed for each subword. The default is true.
		protectedWords (type: string array) - Tokens to protect from being delimited. The default is an empty list.

¹ Token Filter Types are always prefixed in code with `#Microsoft.Azure.Search` such that `ArabicNormalizationTokenFilter` would actually be specified as `#Microsoft.Azure.Search.ArabicNormalizationTokenFilter`. We removed the prefix to reduce the width of the table, but remember to include it in your code.

See also

- [Azure AI Search REST APIs](#)
- [Analyzers in Azure AI Search \(Examples\)](#)
- [Create Index \(REST\)](#)

Create a vector index

09/28/2025

In Azure AI Search, you can use [Create or Update Index \(REST API\)](#) to store vectors in a search index. A vector index is defined by an index schema that has vector fields, nonvector fields, and a vector configuration section.

When you create a vector index, you implicitly create an *embedding space* that serves as the corpus for vector queries. The embedding space consists of all vector fields populated with embeddings from the same embedding model. At query time, the system compares the vector query to the indexed vectors, returning results based on semantic similarity.

To index vectors in Azure AI Search, follow these steps:

- ✓ Start with a basic schema definition.
- ✓ Add vector algorithms and optional compression.
- ✓ Add vector field definitions.
- ✓ Load prevectorized data as a [separate step](#) or use [integrated vectorization](#) for data chunking and embedding during indexing.

This article uses REST for illustration. After you understand the basic workflow, continue with the Azure SDK code samples in the [azure-search-vector-samples](#) repo, which provides guidance on using vectors in test and production code.

Tip

You can also use the Azure portal to [create a vector index](#) and try out integrated data chunking and vectorization.

Prerequisites

- An [Azure AI Search service](#) in any region and on any tier. If you plan to use [integrated vectorization](#) with Azure AI skills and vectorizers, Azure AI Search must be in the same region as the embedding models hosted on Azure AI Vision.
- Your source documents must have [vector embeddings](#) to upload to the index. You can also use [integrated vectorization](#) for this step.
- You should know the dimensions limit of the model that creates the embeddings so that you can assign that limit to the vector field. For `text-embedding-ada-002`, dimensions are

fixed at 1536. For [text-embedding-3-small](#) or [text-embedding-3-large](#), dimensions range from 1 to 1536 and from 1 to 3072, respectively.

- You should know what similarity metric to use. For embedding models on Azure OpenAI, similarity is computed using [cosine](#).
- You should know how to [create an index](#). A schema always includes a field for the document key, fields for search or filters, and other configurations for behaviors needed during indexing and queries.

Limitations

Some search services created before January 2019 can't create a vector index. If this applies to you, create a new service to use vectors.

Prepare documents for indexing

Before indexing, assemble a document payload that includes fields of vector and nonvector data. The document structure must conform to the `fields` collection of index schema.

Make sure your source documents provide the following content:

 Expand table

Content	Description
Unique identifier	A field or a metadata property that uniquely identifies each document. All search indexes require a document key. To satisfy document key requirements, a source document must have one field or property uniquely identifies it in the index. If you're indexing blobs, it might be the <code>metadata_storage_path</code> that uniquely identifies each blob. If you're indexing from a database, it might be primary key. This source field must be mapped to an index field of type <code>Edm.String</code> and <code>key=true</code> in the search index.
Non-vector content	Provide other fields with human-readable content. Human-readable content is useful for the query response and for hybrid queries that include full-text search or semantic ranking in the same request. If you're using a chat completion model, most models like ChatGPT expect human-readable text and don't accept raw vectors as input.
Vector content	A vectorized representation of your nonvector content for use at query time. A vector is an array of single-precision floating point numbers generated by an embedding model. Each vector field contains a model-generated array. There's one embedding per field, where the field is a top-level field (not part of a nested or complex type). For simple integration, we recommend embedding models in Azure OpenAI , such as text-embedding-3 for text documents or the Image Retrieval REST API for images and multimodal embeddings.

If you can use indexers and skillsets, consider [integrated vectorization](#), which encodes

Content	Description
	images and text during indexing. Your field definitions are for vector fields, but incoming source data can be text or images, which are converted to vector arrays during indexing.

Your search index should include fields and content for all of the query scenarios you want to support. Suppose you want to search or filter over product names, versions, metadata, or addresses. In this case, vector similarity search isn't especially helpful. Keyword search, geo-search, or filters that iterate over verbatim content would be a better choice. A search index that's inclusive of both vector and nonvector fields provides maximum flexibility for query construction and response composition.

For a short example of a documents payload that includes vector and nonvector fields, see the [load vector data](#) section of this article.

Start with a basic index

Start with a minimum schema so that you have a definition to work with before adding a vector configuration and vector fields. A simple index might look the following example. For more information about an index schema, see [Create a search index](#).

Notice that the index has a required name, a required document key (`"key": true`), and fields for human-readable content in plain text. It's common to have a human-readable version of whatever content you intend to vectorize. For example, if you have a chunk of text from a PDF file, your index schema should have a field for plain-text chunks and a second field for vectorized chunks.

Here's a basic index with a `"name"`, a `"fields"` collection, and some other constructs for extra configuration:

HTTP

```
POST https://[servicename].search.windows.net/indexes?api-version=[api-version]
{
  "name": "example-index",
  "description": "This is an example index."
  "fields": [
    { "name": "documentId", "type": "Edm.String", "key": true, "retrievable": true, "searchable": true, "filterable": true },
    { "name": "myHumanReadableNameField", "type": "Edm.String", "retrievable": true, "searchable": true, "filterable": false, "sortable": true, "facetable": false },
    { "name": "myHumanReadableContentField", "type": "Edm.String", "retrievable": true, "searchable": true, "filterable": false, "sortable": false, "facetable": false, "analyzer": "en.microsoft" },
  ],
}
```

```
"analyzers": [ ],
"scoringProfiles": [ ],
"suggesters": [ ],
"vectorSearch": [ ]
}
```

Add a vector search configuration

Next, add a `"vectorSearch"` configuration to your schema. It's useful to specify a configuration before field definitions, because the profiles you define here become part of the vector field's definition. In the schema, vector configuration is typically inserted after the fields collection, perhaps after `"analyzers"`, `"scoringProfiles"`, and `"suggesters"`. However, order doesn't matter.

A vector configuration includes:

- `vectorSearch.algorithms` used during indexing to create "nearest neighbor" information among the vector nodes.
- `vectorSearch.compressions` for scalar or binary quantization, oversampling, and reranking with original vectors.
- `vectorSearch.profiles` for specifying multiple combinations of algorithm and compression configurations.

Here are the steps:

1. Use the [Create or Update Index](#) REST API to create the index.
2. Add a `vectorSearch` section in the index that specifies the search algorithms used to create the embedding space.

JSON

```
"vectorSearch": {
    "compressions": [
        {
            "name": "scalar-quantization",
            "kind": "scalarQuantization",
            "scalarQuantizationParameters": {
                "quantizedDataType": "int8"
            },
            "rescoringOptions": {
                "enableRescoring": true,
                "defaultOversampling": 10,
                "rescoreStorageMethod": "preserveOriginals"
            }
        },
        {
            "name": "binary-quantization",
            "kind": "binaryQuantization",
            "binaryQuantizationParameters": {
                "quantizedDataType": "uint8"
            }
        }
    ],
    "profiles": [
        {
            "name": "profile1",
            "algorithms": [
                "cosineSimilarity"
            ],
            "compressions": [
                {
                    "name": "scalar-quantization"
                }
            ],
            "rescoringOptions": {
                "enableRescoring": true
            }
        }
    ]
}
```

```

        "name": "binary-quantization",
        "kind": "binaryQuantization",
        "rescoringOptions": {
            "enableRescoring": true,
            "defaultOversampling": 10,
            "rescoreStorageMethod": "discardOriginals"
        }
    }
],
"algorithms": [
{
    "name": "hnsw-1",
    "kind": "hnsw",
    "hnswParameters": {
        "m": 4,
        "efConstruction": 400,
        "efSearch": 500,
        "metric": "cosine"
    }
},
{
    "name": "hnsw-2",
    "kind": "hnsw",
    "hnswParameters": {
        "m": 8,
        "efConstruction": 800,
        "efSearch": 800,
        "metric": "hamming"
    }
},
{
    "name": "eknn",
    "kind": "exhaustiveKnn",
    "exhaustiveKnnParameters": {
        "metric": "euclidean"
    }
}
]

],
"profiles": [
{
    "name": "vector-profile-hnsw-scalar",
    "compression": "scalar-quantization",
    "algorithm": "hnsw-1"
}
]
}

```

Key points:

- Names for each configuration of compression, algorithm, and profile must be unique for its type within the index.

- `vectorSearch.compressions` can be `scalarQuantization` or `binaryQuantization`. Scalar quantization compresses float values into narrower data types. Binary quantization converts floats into binary 1-bit values.
- `vectorSearch.compressions.rescoringOptions` uses the original, uncompressed vectors to recalculate similarity and rerank the top results returned by the initial search query. The uncompressed vectors exist in the search index even if `stored` is false. This property is optional. Default is true.
- `vectorSearch.compressions.rescoringOptions.defaultOversampling` considers a broader set of potential results to offset the reduction in information from quantization. The formula for potential results consists of the `k` in the query, with an oversampling multiplier. For example, if the query specifies a `k` of 5, and oversampling is 20, the query effectively requests 100 documents for use in reranking, using the original uncompressed vector for that purpose. Only the top `k` reranked results are returned. This property is optional. Default is 4.
- `vectorSearch.compressions.scalarQuantizationParameters.quantizedDataType` must be set to `int8`. This is the only primitive data type supported at this time. This property is optional. Default is `int8`.
- `vectorSearch.algorithms` is either `hnsw` or `exhaustiveKnn`. These are the Approximate Nearest Neighbors (ANN) algorithms used to organize vector content during indexing.
- `vectorSearch.algorithms.m` is the bi-directional link count. Default is 4. The range is 4 to 10. Lower values should return less noise in the results.
- `vectorSearch.algorithms.efConstruction` is the number of nearest neighbors used during indexing. Default is 400. The range is 100 to 1,000.
- `"vectorSearch.algorithms.efSearch"` is the number of nearest neighbors used during search. Default is 500. The range is 100 to 1,000.
- `vectorSearch.algorithms.metric` should be `cosine` if you're using Azure OpenAI, otherwise use the similarity metric associated with the embedding model you're using. Supported values are `cosine`, `dotProduct`, `euclidean`, and `hamming` (used for [indexing binary data](#)).
- `vectorSearch.profiles` add a layer of abstraction for accommodating richer definitions. A profile is defined in `vectorSearch` and referenced by name in each vector field. It's a combination of compression and algorithm configurations. You

assign this property to a vector field, and it determines the fields' algorithm and compression.

Add a vector field to the fields collection

Once you have a vector configuration, you can add a vector field to the fields collection. Recall that the fields collection must include a field for the document key, vector fields, and any other nonvector fields you need for [hybrid search scenarios](#) or chat model completion in [RAG workloads](#).

Vector fields are characterized by [their data type](#), a `dimensions` property based on the embedding model used to output the vectors, and a vector profile that you created in a previous step.

1. Use the [Create or Update Index](#) REST API to create the index and add a vector field to the fields collection.

```
JSON

{
  "name": "example-index",
  "fields": [
    {
      "name": "contentVector",
      "type": "Collection(Edm.Single)",
      "searchable": true,
      "retrievable": false,
      "stored": false,
      "dimensions": 1536,
      "vectorSearchProfile": "vector-profile-1"
    }
  ]
}
```

2. Specify a vector field with the following attributes. You can store one generated embedding per field. For each vector field:

- `type` must be a [vector data type](#). `Collection(Edm.Single)` is the most common for embedding models.
- `dimensions` is the number of dimensions generated by the embedding model. For `text-embedding-ada-002`, it's fixed at 1536. For the `text-embedding-3` model series, there's a range of values. If you're using integrated vectorization and an embedding skill to generate vectors, make sure this property is set to the [same dimensions value](#) used by the embedding skill.
- `vectorSearchProfile` is the name of a profile defined elsewhere in the index.

- `searchable` must be true.
- `retrievable` can be true or false. True returns the raw vectors (1,536 of them) as plain text and consumes storage space. Set to true if you're passing a vector result to a downstream app.
- `stored` can be true or false. It determines whether an extra copy of vectors is stored for retrieval. For more information, see [Reduce vector size](#).
- `filterable`, `facettable`, and `sortable` must be false.

3. Add filterable nonvector fields to the collection, such as `title` with `filterable` set to true, if you want to invoke [prefiltering](#), [postfiltering](#), or [strict postfiltering \(preview\)](#) on the [vector query](#).
4. Add other fields that define the substance and structure of the textual content you're indexing. At a minimum, you need a document key.

You should also add fields that are useful in the query or in its response. The following example shows vector fields for title and content (`titleVector` and `contentVector`) that are equivalent to vectors. It also provides fields for equivalent textual content (`title` and `content`) that are useful for sorting, filtering, and reading in a search result.

The following example shows the fields collection:

HTTP

```
PUT https://my-search-service.search.windows.net/indexes/my-index?api-version=2025-09-01&allowIndexDowntime=true
Content-Type: application/json
api-key: {{admin-api-key}}
{
  "name": "{{index-name}}",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "filterable": true
    },
    {
      "name": "title",
      "type": "Edm.String",
      "searchable": true,
      "filterable": true,
      "sortable": true,
      "retrievable": true
    },
    {
      "name": "titleVector",
      "type": "Collection(Edm.Single)",
      "filterable": false
    }
  ]
}
```

```

        "searchable": true,
        "retrievable": true,
        "stored": true,
        "dimensions": 1536,
        "vectorSearchProfile": "vector-profile-1"
    },
    {
        "name": "content",
        "type": "Edm.String",
        "searchable": true,
        "retrievable": true
    },
    {
        "name": "contentVector",
        "type": "Collection(Edm.Single)",
        "searchable": true,
        "retrievable": false,
        "stored": false,
        "dimensions": 1536,
        "vectorSearchProfile": "vector-profile-1"
    }
],
"vectorSearch": {
    "algorithms": [
        {
            "name": "hnsw-1",
            "kind": "hnsw",
            "hnswParameters": {
                "m": 4,
                "efConstruction": 400,
                "efSearch": 500,
                "metric": "cosine"
            }
        }
    ],
    "profiles": [
        {
            "name": "vector-profile-1",
            "algorithm": "hnsw-1"
        }
    ]
}
}

```

Load vector data for indexing

Content that you provide for indexing must conform to the index schema and include a unique string value for the document key. Prevectorized data is loaded into one or more vector fields, which can coexist with other fields containing nonvector content.

For data ingestion, you can use [push or pull methodologies](#).

Push APIs

Use [Documents - Index](#) to load vector and nonvector data into an index. The push APIs for indexing are identical across all stable and preview versions. Use any of the following APIs to load documents:

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/index?api-version=2025-09-01

{
    "value": [
        {
            "id": "1",
            "title": "Azure App Service",
            "content": "Azure App Service is a fully managed platform for building, deploying, and scaling web apps. You can host web apps, mobile app backends, and RESTful APIs. It supports a variety of programming languages and frameworks, such as .NET, Java, Node.js, Python, and PHP. The service offers built-in auto-scaling and load balancing capabilities. It also provides integration with other Azure services, such as Azure DevOps, GitHub, and Bitbucket.",
            "category": "Web",
            "titleVector": [
                -0.02250031754374504,
                . . .
                ],
            "contentVector": [
                -0.024740582332015038,
                . . .
                ],
            "@search.action": "upload"
        },
        {
            "id": "2",
            "title": "Azure Functions",
            "content": "Azure Functions is a serverless compute service that enables you to run code on-demand without having to manage infrastructure. It allows you to build and deploy event-driven applications that automatically scale with your workload. Functions support various languages, including C#, F#, Node.js, Python, and Java. It offers a variety of triggers and bindings to integrate with other Azure services and external services. You only pay for the compute time you consume.",
            "category": "Compute",
            "titleVector": [
                -0.020159931853413582,
                . . .
                ],
            "contentVector": [
                -0.02780858241021633,
                . . .
                ]
        }
    ]
}
```

```
        ],
        "@search.action": "upload"
    }
    . . .
]
```

Query your index for vector content

For validation purposes, you can query the index using Search Explorer in the Azure portal or a REST API call. Because Azure AI Search can't convert a vector to human-readable text, try to return fields from the same document that provide evidence of the match. For example, if the vector query targets the `titleVector` field, you could select `title` for the search results.

Fields must be attributed as `retrievable` to be included in the results.

Azure portal

- Review the indexes in **Search management > Indexes** to view index size all-up and vector index size. A positive vector index size indicates vectors are present.
- Use [Search Explorer](#) to query an index. Search Explorer has two views: Query view (default) and JSON view.
 - Set **Query options > Hide vector values in search results** for more readable results.
 - [Use the JSON view for vector queries](#). You can paste a JSON definition of the vector query you want to execute. If your index has a [vectorizer assignment](#), you can also use the built-in text-to-vector or image-to-vector conversion. For more information about image search, see [Quickstart: Search for images in Search Explorer](#).
 - Use the default Query view for a quick confirmation that the index contains vectors. The query view is for full-text search. Although you can't use it for vector queries, you can send an empty search (`search=*`) to check for content. The content of all fields, including vector fields, is returned as plain text.

For more information, see [Create a vector query](#).

Update a vector index

To update a vector index, modify the schema and reload documents to populate new fields. APIs for schema updates include [Create or Update Index \(REST\)](#), `CreateOrUpdateIndex` in the Azure SDK for .NET, `create_or_update_index` in the Azure SDK for Python, and similar methods in other Azure SDKs.

For standard guidance on updating an index, see [Update or rebuild an index](#).

Key points include:

- Drop and full index rebuild is often required for updates to and deletion of existing fields.
- You can make the following modifications with no rebuild requirement:
 - Add new fields to a fields collection.
 - Add new vector configurations, assigned to new fields but not existing fields that are already vectorized.
 - Change `retrievable` (values are true or false) on an existing field. Vector fields must be searchable and retrievable, but if you want to disable access to a vector field in situations where drop and rebuild isn't feasible, you can set retrievable to false.

Next steps

As a next step, we recommend [Create a vector query](#).

Code samples in the [azure-search-vector-samples](#) repository demonstrate end-to-end workflows that include schema definition, vectorization, indexing, and queries.

There's demo code for [Python](#), [C#](#), and [JavaScript](#).

Chunk large documents for vector search solutions in Azure AI Search

Partitioning large documents into smaller chunks can help you stay under the maximum token input limits of embedding models. For example, the maximum length of input text for the [Azure OpenAI](#) text-embedding-ada-002 model is 8,191 tokens. Given that each token is around four characters of text for common OpenAI models, this maximum limit is equivalent to around 6,000 words of text. If you're using these models to generate embeddings, it's critical that the input text stays under the limit. Partitioning your content into chunks helps you meet embedding model requirements and prevents data loss due to truncation.

We recommend [integrated vectorization](#) for built-in data chunking and embedding. Integrated vectorization takes a dependency on [built-in indexers](#) and [skillsets](#) that enable text splitting and embeddings generation. If you can't use integrated vectorization, this article describes some alternative approaches for chunking your content.

Common chunking techniques

Chunking is only required if the source documents are too large for the maximum input size imposed by models, but it's also beneficial if content is poorly represented as a single vector. Consider a wiki page that covers a lot of varied sub-topics. The entire page might be small enough to meet model input requirements, but you might get better results if you chunk at a finer grain.

Here are some common chunking techniques, associated with built-in features if you use [indexers](#) and [skills](#).

[] Expand table

Approach	Usage	Built-in functionality
Fixed-size chunks	Define a fixed size that's sufficient for semantically meaningful paragraphs (for example, 200 words or 600 characters) and allows for some overlap (for example, 10-15% of the content) can produce good chunks as input for embedding vector generators.	Text Split skill , splitting by pages (defined by character length)
Variable-sized chunks based on content characteristics	Partition your data based end-of-sentence punctuation marks, end-of-line markers, or using features in the Natural Language Processing (NLP) libraries that detect document structure. Embedded markup, like HTML or Markdown, have heading syntax that can be used to chunk data by sections.	Document Layout skill or Text Split skill , splitting by sentences.

Approach	Usage	Built-in functionality
Custom combinations	Use a combination of fixed and variable sized chunking, or extend an approach. For example, when dealing with large documents, you might use variable-sized chunks, but also append the document title to chunks from the middle of the document to prevent context loss.	None
Document parsing	Indexers can parse larger source documents into smaller search documents for indexing. Strictly speaking, this approach isn't <i>chunking</i> but it can sometimes achieve the same objective.	Index Markdown blobs and files or one-to-many indexing or Index JSON blobs and files

Content overlap considerations

When you chunk data based on fixed size, overlapping a small amount of text between chunks can help maintaining continuity and context. We recommend starting with a chunk size of 512 tokens (approximately 2,000 characters) and an initial overlap of 25%, which equals 128 tokens. This overlap ensures smoother transitions between chunks without excessive duplication.

The optimal overlap may vary depending on your content type and use case. For example, highly structured data may require less overlap, while conversational or narrative text may benefit from more.

Factors for chunking data

When it comes to chunking data, think about these factors:

- Shape and density of your documents. If you need intact text or passages, larger chunks and variable chunking that preserves sentence structure can produce better results.
- User queries: Larger chunks and overlapping strategies help preserve context and semantic richness for queries that target specific information.
- Large Language Models (LLM) have performance guidelines for chunk size. Find a chunk size that works best for all of the models you're using. For instance, if you use models for summarization and embeddings, choose an optimal chunk size that works for both.

How chunking fits into the workflow

If you have large documents, insert a chunking step into indexing and query workflows that breaks up large text. When using [integrated vectorization](#), a default chunking strategy using the [Text Split skill](#) is common. You can also apply a custom chunking strategy using a [custom](#)

skill. See [this code reference](#) for a semantic chunking example using a custom skill. Some external libraries that provide chunking include:

- [LangChain Text Splitters](#)
- [Semantic Kernel TextChunker](#)

Most libraries provide common chunking techniques for fixed size, variable size, or a combination. You can also specify an overlap that duplicates a small amount of content in each chunk for context preservation.

Chunking examples

The following examples demonstrate how chunking strategies are applied to [NASA's Earth at Night e-book](#) PDF file:

- [Text Split skill](#)
- [LangChain](#)
- [Custom skill](#)

Text Split skill example

Integrated data chunking through [Text Split skill](#) is generally available.

This section describes built-in data chunking using a skills-driven approach and [Text Split skill parameters](#).

A sample notebook for this example can be found on the [azure-search-vector-samples](#) repository. Set `textSplitMode` to break up content into smaller chunks:

- `pages` (default). Chunks are made up of multiple sentences.
- `sentences`. Chunks are made up of single sentences. What constitutes a "sentence" is language dependent. In English, standard sentence ending punctuation such as `.` or `!` is used. The language is controlled by the `defaultLanguageCode` parameter.

The `pages` parameter adds extra parameters:

- `maximumPageLength` defines the maximum number of characters ¹ or tokens ² in each chunk. The text splitter avoids breaking up sentences, so the actual character count depends on the content.
- `pageOverlapLength` defines how many characters from the end of the previous page are included at the start of the next page. If set, this must be less than half the maximum page length.

- `maximumPagesToTake` defines how many pages / chunks to take from a document. The default value is 0, which means to take all pages or chunks from the document.

¹ Characters don't align to the definition of a [token](#). The number of tokens measured by the LLM might be different than the character size measured by the Text Split skill with the character fixed-size.

² Token chunking is available in the [2025-08-01-preview](#) and includes extra parameters for specifying a tokenizer and any tokens that shouldn't be split up during chunking.

The following table shows how the choice of parameters affects the total chunk count from the Earth at Night e-book:

[] [Expand table](#)

<code>textSplitMode</code>	<code>maximumPageLength</code>	<code>pageOverlapLength</code>	Total Chunk Count
pages	1000	0	172
pages	1000	200	216
pages	2000	0	85
pages	2000	500	113
pages	5000	0	34
pages	5000	500	38
sentences	N/A	N/A	13361

Using a `textSplitMode` of `pages` results in most chunks having total character counts close to `maximumPageLength`. Chunk character count varies due to differences on where sentence boundaries fall inside the chunk. Chunk token length varies due to differences in the contents of the chunk.

The optimal choice of parameters depends on how the chunks are used. For most applications, it's recommended to start with the following default parameters, when using number of characters:

[] [Expand table](#)

<code>textSplitMode</code>	<code>maximumPageLength</code>	<code>pageOverlapLength</code>
pages	2000	500

LangChain data chunking example

LangChain provides document loaders and text splitters. This example shows you how to load a PDF, get token counts, and set up a text splitter. Getting token counts helps you make an informed decision on chunk sizing.

A sample notebook for this example can be found on the [azure-search-vector-samples](#) repository.

Python

```
from langchain_community.document_loaders import PyPDFLoader
loader = PyPDFLoader("./data/earth_at_night_508.pdf")
pages = loader.load()
print(len(pages))
```

Output indicates 200 documents or pages in the PDF.

To get an estimated token count for these pages, use TikToken.

Python

```
import tiktoken

tokenizer = tiktoken.get_encoding('cl100k_base')
def tiktoken_len(text):
    tokens = tokenizer.encode(
        text,
        disallowed_special=()
    )
    return len(tokens)
tiktoken.encoding_for_model('gpt-4.1-mini')

# create the length function
token_counts = []
for page in pages:
    token_counts.append(tiktoken_len(page.page_content))
min_token_count = min(token_counts)
avg_token_count = int(sum(token_counts) / len(token_counts))
max_token_count = max(token_counts)

# print token counts
print(f"Min: {min_token_count}")
print(f"Avg: {avg_token_count}")
print(f"Max: {max_token_count}")
```

Output indicates that no pages have zero tokens, the average token length per page is 189 tokens, and the maximum token count of any page is 1,583.

Knowing the average and maximum token size gives you insight into setting chunk size. Although you could use the standard recommendation of 2,000 characters with a 500 character overlap, in this case it makes sense to go lower given the token counts of the sample document. In fact, setting an overlap value that's too large can result in no overlap appearing at all.

Python

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
# split documents into text and embeddings

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
    is_separator_regex=False
)

chunks = text_splitter.split_documents(pages)

print(chunks[20])
print(chunks[21])
```

Output for two consecutive chunks shows the text from the first chunk overlapping onto the second chunk. Output is lightly edited for readability.

```
'x Earth at NightForeword\nNASA's Earth at Night explores the brilliance of our planet
when it is in darkness. \n It is a compilation of stories depicting the interactions
between science and \nwonder, and I am pleased to share this visually stunning and
captivating exploration of \nour home planet.\nFrom space, our Earth looks tranquil. The
blue ethereal vastness of the oceans \nharmoniously shares the space with verdant green
land—an undercurrent of gentle-ness and solitude. But spending time gazing at the images
presented in this book, our home planet at night instantly reveals a different reality.
Beautiful, filled with glow-ing communities, natural wonders, and striking illumination,
our world is bustling with activity and life.**\nDarkness is not void of illumination. It
is the contrast, the area between light and'** metadata={'source':
'./data/earth_at_night_508.pdf', 'page': 9}

'''Darkness is not void of illumination. It is the contrast, the area between light and
**\ndark, that is often the most illustrative. Darkness reminds me of where I came from
and where I am now—from a small town in the mountains, to the unique vantage point of the
Nation's capital. Darkness is where dreamers and learners of all ages peer into the
universe and think of questions about themselves and their space in the cosmos. Light is
where they work, where they gather, and take time together.\nNASA's spacefaring
```

```
satellites have compiled an unprecedented record of our \nEarth, and its luminescence in  
darkness, to captivate and spark curiosity. These missions see the contrast between dark  
and light through the lenses of scientific instruments. Our home planet is full of  
complex and dynamic cycles and processes. These soaring observers show us new ways to  
discern the nuances of light created by natural and human-made sources, such as auroras,  
wildfires, cities, phytoplankton, and volcanoes.' metadata={'source':  
'./data/earth_at_night_508.pdf', 'page': 9}
```

Custom skill

A [fixed-sized chunking and embedding generation sample](#) demonstrates both chunking and vector embedding generation using [Azure OpenAI](#) embedding models. This sample uses an [Azure AI Search custom skill](#) in the [Power Skills repo](#) to wrap the chunking step.

See also

- [Understand embeddings in Azure OpenAI in Azure AI Foundry Models](#)
- [Learn how to generate embeddings](#)
- [Tutorial: Explore Azure OpenAI embeddings and document search](#)

Last updated on 10/17/2025

Chunk and vectorize by document layout or structure

Text data chunking strategies play a key role in optimizing RAG responses and performance. By using the **Document Layout** skill, you can chunk content based on document structure, capturing headings and chunking the content body based on semantic coherence, such as paragraphs and sentences. Chunks are processed independently. Because LLMs work with multiple chunks, when those chunks are of higher quality and semantically coherent, the overall relevance of the query is improved.

The Document Layout skill calls the [layout model](#) from Azure Document Intelligence in Foundry Tools. The model articulates content structure in JSON using Markdown syntax (headings and content), with fields for headings and content stored in a search index on Azure AI Search. The searchable content produced from the Document Layout skill is plain text but you can apply integrated vectorization to generate embeddings for any field in your source documents, including images.

In this article, learn how to:

- ✓ Use the Document Layout skill to recognize document structure
- ✓ Use the Text Split skill to constrain chunk size to each markdown section
- ✓ Generate embeddings for each chunk
- ✓ Use index projections to map embeddings to fields in a search index

For illustration purposes, this article uses the [sample health plan PDFs](#) uploaded to Azure Blob Storage and then indexed using the **Import data (new)** wizard.

Prerequisites

- An [indexer-based indexing pipeline](#) with an index that accepts the output. The index must have fields for receiving headings and content.
- An [index projection](#) for one-to-many indexing.
- A [supported data source](#) having text content that you want to chunk.
- A skillset with these two skills:
 - [Document Layout skill](#) that splits documents based on paragraph boundaries. If you use [key-based billing](#), this skill requires Microsoft Foundry to be in the same region as Azure AI Search for AI enrichment. Region requirements are relaxed for keyless billing (preview).

- o Azure OpenAI Embedding skill that generates vector embeddings. This skill *doesn't* have region requirements.

Prepare data files

The raw inputs must be in a [supported data source](#) and the file needs to be a format which [Document Layout skill](#) supports.

- Supported file formats include: PDF, JPEG, JPG, PNG, BMP, TIFF, DOCX, XLSX, PPTX, HTML.
- Supported indexers can be any indexer that can handle the supported file formats. These indexers include [Blob indexers](#), [Microsoft OneLake indexers](#), [File indexers](#).
- Supported regions for the portal experience of this feature include: East US, West Europe, North Central US. If you're setting up your skillset programmatically, you can use any Azure Document Intelligence region that also provides the AI enrichment feature of Azure AI Search. For more information, see [Product availability by region](#).

You can use the Azure portal, REST APIs, or an Azure SDK package to [create a data source](#).

💡 Tip

Upload the [health plan PDF](#) sample files to your supported data source to try out the Document Layout skill and structure-aware chunking on your own search service. The [Import data \(new\) wizard](#) is an easy code-free approach for trying out this skill. Be sure to select the **default parsing mode** to use structure-aware chunking. Otherwise, the [Markdown parsing mode](#) is used.

Create an index for one-to-many indexing

Here's an example payload of a single search document designed around chunks. Whenever you're working with chunks, you need a chunk field and a parent field that identifies the origin of the chunk. In this example, parent fields are the `text_parent_id`. Child fields are the vector and nonvector chunks of the markdown section.

The Document Layout skill outputs headings and content. In this example, `header_1` through `header_3` store document headings, as detected by the skill. Other content, such as paragraphs, is stored in `chunk`. The `text_vector` field is a vector representation of the chunk field content.

You can use the [Import data \(new\)](#) wizard in the Azure portal, REST APIs, or an Azure SDK to [create an index](#). The following index is very similar to what the wizard creates by default. You might have more fields if you add image vectorization.

If you aren't using the wizard, the index must exist on the search service before you create the skillset or run the indexer.

JSON

```
{  
  "name": "my Consolidated Index",  
  "fields": [  
    {  
      "name": "chunk_id",  
      "type": "Edm.String",  
      "searchable": true,  
      "filterable": false,  
      "retrievable": true,  
      "stored": true,  
      "sortable": true,  
      "facetable": false,  
      "key": true,  
      "analyzer": "keyword"  
    },  
    {  
      "name": "text_parent_id",  
      "type": "Edm.String",  
      "searchable": false,  
      "filterable": true,  
      "retrievable": true,  
      "stored": true,  
      "sortable": false,  
      "facetable": false,  
      "key": false  
    },  
    {  
      "name": "chunk",  
      "type": "Edm.String",  
      "searchable": true,  
      "filterable": false,  
      "retrievable": true,  
      "stored": true,  
      "sortable": false,  
      "facetable": false,  
      "key": false  
    },  
    {  
      "name": "title",  
      "type": "Edm.String",  
      "searchable": true,  
      "filterable": false,  
      "retrievable": true,  
      "stored": true,  
      "sortable": true,  
      "facetable": false,  
      "key": false  
    }]
```

```
        "sortable": false,
        "facetable": false,
        "key": false
    },
{
    "name": "header_1",
    "type": "Edm.String",
    "searchable": true,
    "filterable": false,
    "retrievable": true,
    "stored": true,
    "sortable": false,
    "facetable": false,
    "key": false
},
{
    "name": "header_2",
    "type": "Edm.String",
    "searchable": true,
    "filterable": false,
    "retrievable": true,
    "stored": true,
    "sortable": false,
    "facetable": false,
    "key": false
},
{
    "name": "header_3",
    "type": "Edm.String",
    "searchable": true,
    "filterable": false,
    "retrievable": true,
    "stored": true,
    "sortable": false,
    "facetable": false,
    "key": false
},
{
    "name": "text_vector",
    "type": "Collection(Edm.Single)",
    "searchable": true,
    "filterable": false,
    "retrievable": true,
    "stored": true,
    "sortable": false,
    "facetable": false,
    "key": false,
    "dimensions": 1536,
    "stored": false,
    "vectorSearchProfile": "profile"
}
],
"vectorSearch": {
    "profiles": [
        {
            "name": "profile"
        }
    ]
}
```

```
        "name": "profile",
        "algorithm": "algorithm"
    }
],
"algorithms": [
{
    "name": "algorithm",
    "kind": "hnsw"
}
]
}
}
```

Define a skillset for structure-aware chunking and vectorization

This section shows an example of a skillset definition that projects individual markdown sections, chunks, and their vector equivalents as fields in the search index. It uses the [Document Layout skill](#) to detect headings and populate a content field based on semantically coherent paragraphs and sentences in the source document. It uses the [Text Split skill](#) to split the Markdown content into chunks. It uses the [Azure OpenAI Embedding skill](#) to vectorize chunks and any other field for which you want embeddings.

Besides skills, the skillset includes `indexProjections` and `cognitiveServices`:

- `indexProjections` are used for indexes containing chunked documents. The projections specify how parent-child content is mapped to fields in a search index for one-to-many indexing. For more information, see [Define an index projection](#).
- `cognitiveServices` attaches a Foundry resource for billing purposes (the Document Layout skill is available through [Standard pricing ↗](#)).

https

```
POST {endpoint}/skillsets?api-version=2025-09-01
```

```
{
    "name": "my_skillset",
    "description": "A skillset for structure-aware chunking and vectorization with an index projection around markdown section",
    "skills": [
    {
        "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",
        "name": "my_document_intelligence_layout_skill",
        "context": "/document",
        "outputMode": "oneToMany",
        "inputs": [

```

```
{
  "name": "file_data",
  "source": "/document/file_data"
}
],
"outputs": [
  {
    "name": "markdown_document",
    "targetName": "markdownDocument"
  }
],
"markdownHeaderDepth": "h3"
},
{
"@odata.type": "#Microsoft.Skills.Text.SplitSkill",
"name": "my_markdown_section_split_skill",
"description": "A skill that splits text into chunks",
"context": "/document/markdownDocument/*",
"inputs": [
  {
    "name": "text",
    "source": "/document/markdownDocument/*/content",
    "inputs": []
  }
],
"outputs": [
  {
    "name": "textItems",
    "targetName": "pages"
  }
],
"defaultLanguageCode": "en",
"textSplitMode": "pages",
"maximumPageLength": 2000,
"pageOverlapLength": 500,
"unit": "characters"
},
{
"@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
"name": "my_azure_openai_embedding_skill",
"context": "/document/markdownDocument/*/pages/*",
"inputs": [
  {
    "name": "text",
    "source": "/document/markdownDocument/*/pages/*",
    "inputs": []
  }
],
"outputs": [
  {
    "name": "embedding",
    "targetName": "text_vector"
  }
],
"resourceUri": "https://<subdomain>.openai.azure.com",
```

```

    "deploymentId": "text-embedding-3-small",
    "apiKey": "<Azure OpenAI api key>",
    "modelName": "text-embedding-3-small"
  }
],
"cognitiveServices": {
  "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",
  "key": "<Cognitive Services api key>"
},
"indexProjections": {
  "selectors": [
    {
      "targetIndexName": "my Consolidated_index",
      "parentKeyFieldName": "text_parent_id",
      "sourceContext": "/document/markdownDocument/*/pages/*",
      "mappings": [
        {
          "name": "text_vector",
          "source": "/document/markdownDocument/*/pages/*/text_vector"
        },
        {
          "name": "chunk",
          "source": "/document/markdownDocument/*/pages/*"
        },
        {
          "name": "title",
          "source": "/document/title"
        },
        {
          "name": "header_1",
          "source": "/document/markdownDocument/*/sections/h1"
        },
        {
          "name": "header_2",
          "source": "/document/markdownDocument/*/sections/h2"
        },
        {
          "name": "header_3",
          "source": "/document/markdownDocument/*/sections/h3"
        }
      ]
    }
  ],
  "parameters": {
    "projectionMode": "skipIndexingParentDocuments"
  }
}
}

```

Configure and run the indexer

Once you create a data source, index, and skillset, you're ready to [create and run the indexer](#). This step puts the pipeline into execution.

When using the [Document Layout skill](#), make sure to set the following parameters on the indexer definition:

- The `allowSkillsetToReadFileData` parameter should be set to `true`.
- the `parsingMode` parameter should be set to `default`.

`outputFieldMappings` don't need to be set in this scenario because `indexProjections` handle the source field to search field associations. Index projections handle field associations for the Document Layout skill and also regular chunking with the split skill for imported and vectorized data workloads. Output field mappings are still necessary for transformations or complex data mappings with functions which apply in other cases. However, for n-chunks per document, index projections handle this functionality natively.

Here's an example of an indexer creation request.

https

```
POST {endpoint}/indexers?api-version=2025-09-01

{
  "name": "my_indexer",
  "dataSourceName": "my_blob_datasource",
  "targetIndexName": "my_consolidated_index",
  "skillsetName": "my_skillset",
  "parameters": {
    "batchSize": 1,
    "configuration": {
      "dataToExtract": "contentAndMetadata",
      "parsingMode": "default",
      "allowSkillsetToReadFileData": true
    }
  },
  "fieldMappings": [
    {
      "sourceFieldName": "metadata_storage_path",
      "targetFieldName": "title"
    }
  ],
  "outputFieldMappings": []
}
```

When you send the request to the search service, the indexer runs.

Verify results

You can query your search index after processing concludes to test your solution.

To check the results, run a query against the index. Use [Search Explorer](#) as a search client, or any tool that sends HTTP requests. The following query selects fields that contain the output of markdown section nonvector content and its vector.

For Search Explorer, you can copy just the JSON and paste it into the JSON view for query execution.

HTTP

```
POST /indexes/[index name]/docs/search?api-version=[api-version]
{
  "search": "copay for in-network providers",
  "count": true,
  "searchMode": "all",
  "vectorQueries": [
    {
      "kind": "text",
      "text": "*",
      "fields": "text_vector,image_vector"
    }
  ],
  "queryType": "semantic",
  "semanticConfiguration": "healthplan-doc-layout-test-semantic-configuration",
  "captions": "extractive",
  "answers": "extractive|count-3",
  "queryLanguage": "en-us",
  "select": "header_1, header_2, header_3"
}
```

If you used the health plan PDFs to test this skill, Search Explorer results for the example query should look similar to the results in the following screenshot.

- The query is a [hybrid query](#) over text and vectors, so you see a `@search.rerankerScore` and results are ranked by that score. `searchMode=all` means that *all* query terms must be considered for a match (the default is *any*).
- The query uses semantic ranking, so you see `captions` (it also has `answers`, but those aren't shown in the screenshot). The results are the most semantically relevant to the query input, as determined by the [semantic ranker](#).
- The `select` statement (not shown in the screenshot) specifies the header fields that the Document Layout skill detects and populates. You can add more fields to the select clause to inspect the content of chunks, title, or any other human readable field.

JSON query editor

```
2   "search": "copay for in-network providers",
3   "count": true,
4   "searchMode": "all",
5   "vectorQueries": [
6     {
7       "kind": "text",
8       "text": "copay for in-network providers",
9       "fields": "text_vector,image_vector"
```

Search

Results

56	"@search.rerankerScore": 3.2406420707702637,
57	"@search.captions": [
58	{
59	"text": "When using Northwind Health Plus, you may be responsible for a copayment (or copay) for cer
60	"highlights": ""
61	}
62],
63	"header_1": "Contoso Electronics",
64	"header_2": "IMPORTANT PLAN INFORMATION",
65	"header_3": "Copayments (Copays) IMPORTANT PLAN INFORMATION: Copayments (Copays)"
66	},
67	{
68	"@search.score": 0.01666666753590107,
69	"@search.rerankerScore": 2.999603033065796,
70	"@search.captions": [
71	{
72	"text": "Choosing in-network providers have agreed to accept a discounted rate on se
73	"highlights": "Choosing in-network providers have agreed to accept a discounted
74	}
75],
76	"header_1": "Contoso Electronics",
77	"header_2": "HOW PROVIDERS AFFECT YOUR COSTS",
78	"header_3": "In-Network Providers"
79	},
80	

See also

- [Create or update a skill set.](#)
- [Create a data source](#)
- [Define an index projection](#)
- [Attach a Foundry resource](#)
- [Document Layout skill](#)
- [Text Split skill](#)
- [Azure OpenAI Embedding skill](#)
- [Create indexer \(REST\)](#)
- [Search Explorer](#)

Generate embeddings for search queries and documents

Azure AI Search doesn't host embedding models, so you're responsible for creating vectors for query inputs and outputs. Choose one of the following approaches:

 Expand table

Approach	Description
Integrated vectorization	Use built-in data chunking and vectorization in Azure AI Search. This approach takes a dependency on indexers, skillsets, and built-in or custom skills that point to external embedding models, such as those in Microsoft Foundry.
Manual vectorization	Manage data chunking and vectorization yourself. For indexing, you push prevectorized documents into vector fields in a search index. For querying, you provide precomputed vectors to the search engine. For demos of this approach, see the azure-search-vector-samples GitHub repository.

We recommend integrated vectorization for most scenarios. Although you can use any supported embedding model, this article uses Azure OpenAI models for illustration.

How embedding models are used in vector queries

Embedding models generate vectors for both query inputs and query outputs. Query inputs include:

- **Text or images that are converted to vectors during query processing.** As part of integrated vectorization, a [vectorizer](#) performs this task.
- **Precomputed vectors.** You can generate these vectors by passing the query input to an embedding model of your choice. To avoid [rate limiting](#), implement retry logic in your workload. Our [Python demo](#) uses [tenacity](#).

Based on the query input, the search engine retrieves matching documents from your search index. These documents are the query outputs.

Your search index must already contain documents with one or more vector fields populated by embeddings. You can create these embeddings through integrated or manual vectorization. To ensure accurate results, use the same embedding model for indexing and querying.

Tips for embedding model integration

- **Identify use cases.** Evaluate specific use cases where embedding model integration for vector search features adds value to your search solution. Examples include [multimodal search](#) or matching image content with text content, multilingual search, and similarity search.
- **Design a chunking strategy.** Embedding models have limits on the number of tokens they accept, so [data chunking](#) is necessary for large files.
- **Optimize cost and performance.** Vector search is resource intensive and subject to maximum limits, so vectorize only the fields that contain semantic meaning. [Reduce vector size](#) to store more vectors for the same price.
- **Choose the right embedding model.** Select a model for your use case, such as word embeddings for text-based searches or image embeddings for visual searches. Consider pretrained models, such as text-embedding-ada-002 from OpenAI or the Image Retrieval REST API from [Azure Vision in Foundry Tools](#).
- **Normalize vector lengths.** To improve the accuracy and performance of similarity search, normalize vector lengths before you store them in a search index. Most pretrained models are already normalized.
- **Fine-tune the model.** If needed, fine-tune the model on your domain-specific data to improve its performance and relevance to your search application.
- **Test and iterate.** Continuously test and refine the embedding model integration to achieve your desired search performance and user satisfaction.

Create resources in the same region

Although integrated vectorization with Azure OpenAI embedding models doesn't require resources to be in the same region, using the same region can improve performance and reduce latency.

To use the same region for your resources:

1. Check the [regional availability of text embedding models](#).
2. Check the [regional availability of Azure AI Search](#).
3. Create an Azure OpenAI resource and Azure AI Search service in the same region.

Tip

Want to use [semantic ranking](#) for [hybrid queries](#) or a machine learning model in a [custom skill](#) for [AI enrichment](#)? Choose an Azure AI Search region that provides those features.

Choose an embedding model in Foundry

When you add knowledge to an agent workflow in the [Foundry portal](#), you have the option of creating a search index. A wizard guides you through the steps.

One step involves selecting an embedding model to vectorize your plain text content. The following models are supported:

- text-embedding-3-small
- text-embedding-3-large
- text-embedding-ada-002
- Cohere-embed-v3-english
- Cohere-embed-v3-multilingual

Your model must already be deployed, and you must have permission to access it. For more information, see [Deployment overview for Foundry Models](#).

Generate an embedding for an improvised query

If you don't want to use integrated vectorization, you can manually generate an embedding and paste it into the `vectorQueries.vector` property of a vector query. For more information, see [Create a vector query in Azure AI Search](#).

The following examples assume the text-embedding-ada-002 model. Replace `YOUR-API-KEY` and `YOUR-OPENAI-RESOURCE` with your Azure OpenAI resource details.

.NET

C#

```
using System;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json;

class Program
{
```

```
static async Task Main(string[] args)
{
    var apiKey = "YOUR-API-KEY";
    var apiBase = "https://YOUR-OPENAI-RESOURCE.openai.azure.com";
    var apiVersion = "2024-02-01";
    var engine = "text-embedding-ada-002";

    var client = new HttpClient();
    client.DefaultRequestHeaders.Add("Authorization", $"Bearer {apiKey}");

    var requestBody = new
    {
        input = "How do I use C# in VS Code?"
    };

    var response = await client.PostAsync(
        $"{apiBase}/openai/deployments/{engine}/embeddings?api-version={apiVersion}",
        new StringContent(JsonConvert.SerializeObject(requestBody),
Encoding.UTF8, "application/json")
    );

    var responseBody = await response.Content.ReadAsStringAsync();
    Console.WriteLine(responseBody);
}
```

The output is a vector array of 1,536 dimensions.

Related content

- [Understand embeddings in Azure OpenAI in Foundry Models](#)
- [Generate embeddings with Azure OpenAI](#)
- [Tutorial: Explore Azure OpenAI embeddings and document search](#)
- [Tutorial: Choose a model \(RAG solutions in Azure AI Search\)](#)

Last updated on 11/18/2025

Set up integrated vectorization in Azure AI Search using REST

In this article, you learn how to use a skillset to chunk and vectorize content from a [supported data source](#). The skillset calls the [Text Split skill](#) or [Document Layout skill](#) for chunking and an embedding skill that's attached to a [supported embedding model](#) for chunk vectorization. You also learn how to store the chunked and vectorized content in a [vector index](#).

This article describes the end-to-end workflow for [integrated vectorization](#) using REST. For portal-based instructions, see [Quickstart: Vectorize text and images in the Azure portal](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An [Azure AI Search service](#). We recommend the Basic tier or higher.
- A [supported data source](#).
- A [supported embedding model](#).
- Completion of [Quickstart: Connect without keys](#) and [Configure a system-assigned managed identity](#). Although you can use key-based authentication for data plane operations, this article assumes [roles and managed identities](#), which are more secure.
- [Visual Studio Code](#) with a [REST client](#).

Supported data sources

Integrated vectorization works with [all supported data sources](#). However, this article focuses on the most commonly used data sources, which are described in the following table.

 Expand table

Supported data source	Description
Azure Blob Storage	This data source works with blobs and tables. You must use a standard performance (general-purpose v2) account. Access tiers can be hot, cool, or cold.
Azure Data Lake Storage (ADLS) Gen2	This is an Azure Storage account with a hierarchical namespace enabled. To confirm that you have Data Lake Storage, check the Properties tab on the Overview page.

Supported data source	Description
	 Screenshot of an Azure Data Lake Storage account in the Azure portal.
Microsoft OneLake	This data source connects to OneLake files and shortcuts.

Supported embedding models

For integrated vectorization, use one of the following embedding models on an Azure AI platform. Deployment instructions are provided in a [later section](#).

 Expand table

Provider	Supported models
Azure OpenAI in Foundry Models ^{1, 2}	text-embedding-ada-002 text-embedding-3-small text-embedding-3-large
Microsoft Foundry resource ³	For text and images: Azure Vision multimodal ⁴

¹ The endpoint of your Azure OpenAI resource must have a [custom subdomain](#), such as <https://my-unique-name.openai.azure.com>. If you created your resource in the [Azure portal](#) , this subdomain was automatically generated during resource setup.

² Azure OpenAI resources (with access to embedding models) that were created in the [Foundry portal](#)  aren't supported. You must create an Azure OpenAI resource in the Azure portal.

³ For billing purposes, you must [attach your Foundry resource](#) to the skillset in your Azure AI Search service. Unless you use a [keyless connection \(preview\)](#) to create the skillset, both resources must be in the same region.

⁴ The Azure Vision multimodal embedding model is available in [select regions](#).

Role-based access

You can use Microsoft Entra ID with role assignments or key-based authentication with full-access connection strings. For Azure AI Search connections to other resources, we recommend role assignments.

To configure role-based access for integrated vectorization:

1. On your search service, [enable roles](#) and [configure a system-assigned managed identity](#).

2. On your data source platform and embedding model provider, create role assignments that allow your search service to access data and models. See [Prepare your data](#) and [Prepare your embedding model](#).

(!) Note

Free search services support role-based connections to Azure AI Search. However, they don't support managed identities on outbound connections to Azure Storage or Azure Vision. This lack of support requires that you use key-based authentication on connections between free search services and other Azure resources.

For more secure connections, use the Basic tier or higher. You can then enable roles and configure a managed identity for authorized access.

Get connection information for Azure AI Search

In this section, you retrieve the endpoint and Microsoft Entra token for your Azure AI Search service. Both values are necessary to establish connections in REST requests.

💡 Tip

The following steps assume that you're using [role-based access](#) for proof-of-concept testing. If you want to use integrated vectorization for app development, see [Connect your app to Azure AI Search using identities](#).

1. Sign in to the [Azure portal](#) and select your Azure AI Search service.
2. To obtain your search endpoint, copy the URL on the **Overview** page. An example search endpoint is `https://my-service.search.windows.net`.
3. To obtain your Microsoft Entra token, run the following command on your local system. This step requires completion of [Quickstart: Connect without keys](#).

Azure CLI

```
az account get-access-token --scope https://search.azure.com/.default --query accessToken --output tsv
```

Prepare your data

In this section, you prepare your data for integrated vectorization by uploading files to a [supported data source](#), assigning roles, and obtaining connection information.

Azure Blob Storage

1. Sign in to the [Azure portal](#) and select your Azure Storage account.
2. From the left pane, select **Data storage > Containers**.
3. Create a container or select an existing container, and then upload your files to the container.
4. To assign roles:
 - a. From the left pane, select **Access Control (IAM)**.
 - b. Select **Add > Add role assignment**.
 - c. Under **Job function roles**, select [Storage Blob Data Reader](#), and then select **Next**.
 - d. Under **Members**, select **Managed identity**, and then select **Select members**.
 - e. Select your subscription and the managed identity of your search service.
5. To obtain a connection string:
 - a. From the left pane, select **Security + networking > Access keys**.
 - b. Copy either connection string, which you specify later in [Set variables](#).
6. (Optional) Synchronize deletions in your container with deletions in the search index.
To configure your indexer for deletion detection:
 - a. [Enable soft delete](#) on your storage account. If you're using [native soft delete](#), the next step isn't required.
 - b. [Add custom metadata](#) that an indexer can scan to determine which blobs are marked for deletion. Give your custom property a descriptive name. For example, you can name the property "IsDeleted" and set it to false. Repeat this step for every blob in the container. When you want to delete the blob, change the property to true. For more information, see [Change and delete detection when indexing from Azure Storage](#).

Prepare your embedding model

In this section, you prepare your Azure AI resource for integrated vectorization by assigning roles, obtaining an endpoint, and deploying a [supported embedding model](#).

Azure OpenAI

Azure AI Search supports text-embedding-ada-002, text-embedding-3-small, and text-embedding-3-large. Internally, Azure AI Search calls the [Azure OpenAI Embedding skill](#) to connect to Azure OpenAI.

1. Sign in to the [Azure portal](#) and select your Azure OpenAI resource.
2. To assign roles:
 - a. From the left pane, select **Access control (IAM)**.
 - b. Select **Add > Add role assignment**.
 - c. Under **Job function roles**, select [Cognitive Services OpenAI User](#), and then select **Next**.
 - d. Under **Members**, select **Managed identity**, and then select **Select members**.
 - e. Select your subscription and the managed identity of your search service.
3. To obtain an endpoint:
 - a. From the left pane, select **Resource Management > Keys and Endpoint**.
 - b. Copy the endpoint for your Azure OpenAI resource. You specify this URL later in [Set variables](#).
4. To deploy an embedding model:
 - a. Sign in to the [Foundry portal](#) and select your Azure OpenAI resource.
 - b. Deploy a [supported embedding model](#).
 - c. Copy the deployment and model names, which you specify later in [Set variables](#).
The deployment name is the custom name you chose, while the model name is the model you deployed, such as `text-embedding-ada-002`.

Set variables

In this section, you specify the connection information for your Azure AI Search service, your supported data source, and your [supported embedding model](#).

1. In Visual Studio Code, paste the following placeholders into your `.rest` or `.http` file.

HTTP

```
@baseUrl = PUT-YOUR-SEARCH-SERVICE-URL-HERE  
@token = PUT-YOUR-MICROSOFT-ENTRA-TOKEN-HERE
```

2. Replace `@baseUrl` with the search endpoint and `@token` with the Microsoft Entra token you obtained in [Get connection information for Azure AI Search](#).
3. Depending on your data source, add the following variables.

[+] [Expand table](#)

Data source	Variables	Enter this information
Azure Blob Storage	<code>@storageConnectionString</code> and <code>@blobContainer</code>	The connection string and the name of the container you created in Prepare your data .
ADLS Gen2	<code>@storageConnectionString</code> and <code>@blobContainer</code>	The connection string and the name of the container you created in Prepare your data .
OneLake	<code>@workspaceId</code> and <code>@lakehouseId</code>	The workspace and lakehouse IDs you obtained in Prepare your data .

4. Depending on your embedding model provider, add the following variables.

[+] [Expand table](#)

Embedding model provider	Variables	Enter this information
Azure OpenAI	<code>@aoaiEndpoint</code> , <code>@aoaiDeploymentName</code> , and <code>@aoaimodelName</code>	The endpoint, deployment name, and model name you obtained in Prepare your embedding model .
Azure Vision	<code>@AiFoundryEndpoint</code>	The endpoint you obtained in Prepare your embedding model .

5. To verify the variables, send the following request.

HTTP

```
### List existing indexes by name
GET {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
Authorization: Bearer {{token}}
```

A response should appear in an adjacent pane. If you have existing indexes, they're listed. Otherwise, the list is empty. If the HTTP code is `200 OK`, you're ready to proceed.

Connect to your data

In this section, you connect to a [supported data source](#) for indexer-based indexing. An [indexer](#) in Azure AI Search requires a data source that specifies the type, credentials, and container.

1. Use [Create Data Source](#) to define a data source that provides connection information during indexing.

HTTP

```
### Create a data source
POST {{baseUrl}}/datasources?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
Authorization: Bearer {{token}}


{
  "name": "my-data-source",
  "type": "azureblob",
  "subtype": null,
  "credentials": {
    "connectionString": "{{storageConnectionString}}"
  },
  "container": {
    "name": "{{blobContainer}}",
    "query": null
  },
  "dataChangeDetectionPolicy": null,
  "dataDeletionDetectionPolicy": null
}
```

2. Set `type` to your data source: `azureblob` or `adlsgen2`.
3. To create the data source, select **Send request**.
4. If you're using OneLake, set `credentials.connectionString` to `ResourceId={{workspaceId}}` and `container.name` to `{{lakehouseId}}`.

Create a skillset

In this section, you create a [skillset](#) that calls a built-in skill to chunk your content and an embedding skill to create vector representations of the chunks. The skillset is executed during indexing in a [later section](#).

Call a built-in skill to chunk your content

Partitioning your content into chunks helps you meet the requirements of your embedding model and prevents data loss due to truncation. For more information about chunking, see [Chunk large documents for vector search solutions](#).

For built-in data chunking, Azure AI Search offers the [Text Split skill](#) and [Document Layout skill](#). The Text Split skill breaks text into sentences or pages of a particular length, while the Document Layout skill breaks content based on paragraph boundaries.

1. Use [Create Skillset](#) to define a skillset.

HTTP

```
### Create a skillset
POST {{baseUrl}}/skillsets?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
Authorization: Bearer {{token}}


{
  "name": "my-skillset",
  "skills": []
}
```

2. In the `skills` array, call the Text Split skill or Document Layout skill. You can paste one of the following definitions.

HTTP

```
"skills": [
  {
    "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
    "name": "my-text-split-skill",
    "textSplitMode": "pages",
    "maximumPageLength": 2000,
    "pageOverlapLength": 500,
    "maximumPagesToTake": 0,
    "unit": "characters",
    "defaultLanguageCode": "en",
    "inputs": [
      {
        "name": "text",
        "source": "/document/text",
        "inputs": []
    }
]
```

```
        }
    ],
    "outputs": [
    {
        "name": "textItems"
    }
]
},
{
    "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",
    "name": "my-document-layout-skill",
    "context": "/document",
    "outputMode": "oneToMany",
    "markdownHeaderDepth": "h3",
    "inputs": [
    {
        "name": "file_data",
        "source": "/document/file_data"
    }
],
    "outputs": [
    {
        "name": "markdown_document"
    }
]
}
]
```

ⓘ Note

The Document Layout skill is in public preview. If you want to call this skill, use a preview API, such as [2025-03-01-preview](#).

Call an embedding skill to vectorize the chunks

To vectorize your chunked content, the skillset needs an embedding skill that points to a [supported embedding model](#).

1. After the built-in chunking skill in the `skills` array, call the [Azure OpenAI Embedding skill](#) or [Azure Vision skill](#). You can paste one of the following definitions.

HTTP

```
{
    "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
    "resourceUri": "{{aoaiEndpoint}}",
    "deploymentId": "{{aoaiDeploymentName}}",
    "modelName": "{{aoai modelName}}",
    "dimensions": 1536,
```

```
"inputs": [
  {
    "name": "text",
    "source": "/document/text"
  }
],
"outputs": [
  {
    "name": "embedding"
  }
]
},
{
  "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
  "context": "/document",
  "modelVersion": "2023-04-15",
  "inputs": [
    {
      "name": "url",
      "source": "/document/metadata_storage_path"
    },
    {
      "name": "queryString",
      "source": "/document/metadata_storage_sas_token"
    }
  ],
  "outputs": [
    {
      "name": "vector"
    }
  ]
}
```

ⓘ Note

The Azure Vision multimodal embeddings skill is in public preview. If you want to call this skill, use the latest preview API.

2. If you're using the Azure OpenAI Embedding skill, set `dimensions` to the [number of embeddings generated by your embedding model](#).
3. If you're using the Azure Vision multimodal embeddings skill, [attach your Foundry resource](#) after the `skills` array. This attachment is for billing purposes.

HTTP

```
"skills": [ ... ],
"cognitiveServices": {
  "@odata.type": "#Microsoft.Azure.Search.AIServicesByIdentity",
```

```
        "subdomainUrl": "{{AiFoundryEndpoint}}"  
    }  
}
```

4. To create the skillset, select **Send request**.

Create a vector index

In this section, you set up physical data structures on your Azure AI Search service by creating a [vector index](#). The schema of a vector index requires the following:

- Name
- Key field (string)
- One or more vector fields
- Vector configuration

Vector fields store numerical representations of your chunked data. They must be searchable and retrievable, but they can't be filterable, facetable, or sortable. They also can't have analyzers, normalizers, or synonym map assignments.

In addition to vector fields, the sample index in the following steps contains nonvector fields for human-readable content. It's common to include plain-text equivalents of the content you want to vectorize. For more information, see [Create a vector index](#).

1. Use [Create Index](#) to define the schema of a vector index.

HTTP

```
### Create a vector index  
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1  
Content-Type: application/json  
Authorization: Bearer {{token}}  
  
{  
    "name": "my-vector-index",  
    "fields": [],  
    "vectorSearch": []  
}
```

2. Add a [vector search configuration](#) to the `vectorSearch` section.

HTTP

```
"vectorSearch": {  
    "algorithms": [  
        {  
            "name": "hnsw-algorithm",  
            "kind": "hnsw",
```

```

    "hnswParameters": {
        "m": 4,
        "efConstruction": 400,
        "efSearch": 100,
        "metric": "cosine"
    }
},
],
"profiles": [
{
    "name": "vector-profile-hnsw",
    "algorithm": "hnsw-algorithm",
}
]
}

```

`vectorSearch.algorithms` specifies the algorithm used for indexing and querying vector fields, while `vectorSearch.profiles` links the algorithm configuration to a profile you can assign to vector fields.

3. Depending on your embedding model, update `vectorSearch.algorithms.metric`. Valid values for distance metrics are `cosine`, `dotproduct`, `euclidean`, and `hamming`.
4. Add fields to the `fields` arrays. Include a key field for document identification, nonvector fields for human-readable content, and vector fields for embeddings.

HTTP

```

"fields": [
{
    "name": "id",
    "type": "Edm.String",
    "key": true,
    "filterable": true
},
{
    "name": "title",
    "type": "Edm.String",
    "searchable": true,
    "filterable": true,
    "sortable": true,
    "retrievable": true
},
{
    "name": "titleVector",
    "type": "Collection(Edm.Single)",
    "searchable": true,
    "retrievable": false,
    "stored": true,
    "dimensions": 1536,
    "vectorSearchProfile": "vector-profile-hnsw"
}
]
}

```

```

},
{
  "name": "content",
  "type": "Edm.String",
  "searchable": true,
  "retrievable": true
},
{
  "name": "contentVector",
  "type": "Collection(Edm.Single)",
  "searchable": true,
  "retrievable": false,
  "stored": false,
  "dimensions": 1536,
  "vectorSearchProfile": "vector-profile-hnsw"
}
]

```

5. Depending on your embedding skill, set `dimensions` for each vector field to the following value.

[] [Expand table](#)

Embedding skill	Enter this value
Azure OpenAI	The number of embeddings generated by your embedding model .
Azure Vision	<code>1024</code>

Add a vectorizer to the index

In this section, you enable vectorization at query time by [defining a vectorizer](#) in your index. The vectorizer uses the embedding model that indexes your data to decode a search string or image into a vector for vector search.

1. Add the [Azure OpenAI vectorizer](#) or [Azure Vision vectorizer](#) after `vectorSearch.profiles`. You can paste one of the following definitions.

HTTP

```

"profiles": [ ... ],
"vectorizers": [
  {
    "name": "my-openai-vectorizer",
    "kind": "azureOpenAI",
    "azureOpenAIParameters": {
      "resourceUri": "{{aoaiEndpoint}}",
      "deploymentId": "{{aoaiDeploymentName}}",
      "modelName": "{{aoai modelName}}"
    }
  }
]

```

```
        }
    },
{
    "name": "my-ai-services-vision-vectorizer",
    "kind": "aiServicesVision",
    "aiServicesVisionParameters": {
        "resourceUri": "{{AiFoundryEndpoint}}",
        "modelVersion": "2023-04-15"
    }
}
]
```

ⓘ Note

The Azure Vision vectorizer is in public preview. If you want to call this vectorizer, use a preview API, such as [2025-03-01-preview](#).

2. Specify your vectorizer in `vectorSearch.profiles`.

HTTP

```
"profiles": [
{
    "name": "vector-profile-hnsw",
    "algorithm": "hnsw-algorithm",
    "vectorizer": "my-openai-vectorizer"
}]
```

3. To create the vector index, select **Send request**.

Create an indexer

In this section, you create an [indexer](#) to drive the entire vectorization pipeline, from data retrieval to skillset execution to indexing. We recommend that you [run the indexer on a schedule](#) to process changes or documents that were missed due to throttling.

1. Use [Create Indexer](#) to define an indexer that executes the vectorization pipeline.

HTTP

```
### Create an indexer
POST {{baseUrl}}/indexers?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
Authorization: Bearer {{token}}
{
}
```

```
"name": "my-indexer",
"dataSourceName": "my-data-source",
"targetIndexName": "my-vector-index",
"skillsetName": "my-skillset",
"schedule": {
    "interval": "PT2H"
},
"parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null
}
}
```

2. To create the indexer, select **Send request**.

Run a vector query to confirm indexing

In this section, you verify that your content was successfully indexed by [creating a vector query](#). Because you configured a vectorizer in a [previous section](#), the search engine can decode plain text or an image into a vector for query execution.

1. Use [Documents - Search Post](#) to define a query that's vectorized at query time.

HTTP

```
### Run a vector query
POST {{baseUrl}}/indexes('my-vector-index')/docs/search.post.search?api-
version=2025-09-01 HTTP/1.1
Content-Type: application/json
Authorization: Bearer {{token}}


{
  "count": true,
  "select": "title, content",
  "vectorQueries": [
    {
      "kind": "text",
      "text": "a sample text string for integrated vectorization",
      "fields": "titleVector, contentVector",
      "k": "3"
    }
  ]
}
```

ⓘ Note

The Azure Vision vectorizer is in public preview. If you previously called this vectorizer, use a preview API, such as [2025-03-01-preview](#).

For queries that invoke integrated vectorization, `kind` must be set to `text`, and `text` must specify a text string. This string is passed to the vectorizer assigned to the vector field. For more information, see [Query with integrated vectorization](#).

2. To run the vector query, select **Send request**.

Related content

- [Integrated vectorization in Azure AI Search](#)
- [Quickstart: Vectorize text and images in the Azure portal](#)
- [Python sample for integrated vectorization ↗](#)

Last updated on 11/18/2025

Use embedding models from the Microsoft Foundry model catalog for integrated vectorization

Important

This feature is in public preview under [Supplemental Terms of Use](#). The latest preview version of [Skillsets - Create Or Update \(REST API\)](#) supports this feature.

In this article, you learn how to access embedding models from the [Microsoft Foundry model catalog](#) for vector conversions during indexing and query execution in Azure AI Search.

The workflow requires that you deploy a model from the catalog, which includes embedding models from Microsoft and other companies. Deploying a model is billable according to the billing structure of each provider.

After the model is deployed, you can use it with the [AML skill](#) for integrated vectorization during indexing or with the [Microsoft Foundry model catalog vectorizer](#) for queries.

Tip

Use the [Import data \(new\) wizard](#) to generate a skillset that includes an AML skill for deployed embedding models on Foundry. AML skill definition for inputs, outputs, and mappings are generated by the wizard, which gives you an easy way to test a model before writing any code.

Prerequisites

- An [Azure AI Search service](#) in any region and on any pricing tier.
- A [Microsoft Foundry hub-based project](#).

Supported embedding models

Supported embedding models from the model catalog vary by usage method:

- For the latest list of models supported programmatically, see the [AML skill](#) and [Microsoft Foundry model catalog vectorizer](#) references.

- For the latest list of models supported in the Azure portal, see [Quickstart: Vector search in the Azure portal](#) and [Quickstart: Multimodal search in the Azure portal](#).

Deploy an embedding model from the model catalog

1. Follow [these instructions](#) to deploy a supported model to your project.
2. Make a note of the target URI, key, and model name. You need these values for the vectorizer definition in a search index and for the skillset that calls the model endpoints during indexing.

If you prefer [token authentication](#) to key-based authentication, you only need to copy the URI and model name. However, make a note of the region to which the model is deployed.
3. Configure a search index and indexer to use the deployed model.
 - To use the model during indexing, see [How to use integrated vectorization](#). Be sure to use the [AML skill](#), not the [Azure OpenAI Embedding skill](#). The next section describes the skill configuration.
 - To use the model as a vectorizer at query time, see [Configure a vectorizer](#). Be sure to use the [Microsoft Foundry model catalog vectorizer](#) for this step.

Sample AML skill payload

When you deploy embedding models from the model catalog, you connect to them using the [AML skill](#) in Azure AI Search for indexing workloads.

This section describes the AML skill definition and index mappings. It includes a sample payload that's already configured to work with its corresponding deployed endpoint. For more information, see [Skill context and input annotation language](#).

Cohere embedding models

This AML skill payload works with the following embedding models:

- Cohere-embed-v3-english
- Cohere-embed-v3-multilingual
- Cohere-embed-v4

It assumes that you're chunking your content using the Text Split skill and therefore your text to be vectorized is in the `/document/pages/*` path. If your text comes from a different path, update all references to the `/document/pages/*` path accordingly.

You must add the `/v1/embed` path onto the end of the URL that you copied from your Foundry deployment. You might also change the values for the `input_type`, `truncate`, and `embedding_types` inputs to better fit your use case. For more information on the available options, review the [Cohere Embed API reference](#).

The URI and key are generated when you deploy the model from the catalog. For more information about these values, see [How to deploy Cohere Embed models with Foundry](#).

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",  
    "context": "/document/pages/*",  
    "uri": "https://cohere-embed-v3-multilingual-  
    hin.eastus.models.ai.azure.com/v1/embed",  
    "key": "aaaaaaaa-0b0b-1c1c-2d2d-333333333333",  
    "inputs": [  
        {  
            "name": "texts",  
            "source": "[$(/document/pages/*)]"  
        },  
        {  
            "name": "input_type",  
            "source": "'search_document'"  
        },  
        {  
            "name": "truncate",  
            "source": "'NONE'"  
        },  
        {  
            "name": "embedding_types",  
            "source": "[ 'float']"  
        }  
    "outputs": [  
        {  
            "name": "embeddings",  
            "targetName": "aml_vector_data"  
        }  
}
```

In addition, the output of the Cohere model isn't the embeddings array directly, but rather a JSON object that contains it. You need to select it appropriately when mapping it to the index

definition via `indexProjections` or `outputFieldMappings`. Here's a sample `indexProjections` payload that would allow you to do implement this mapping.

If you selected a different `embedding_types` in your skill definition, change `float` in the `source` path to the type you selected.

JSON

```
"indexProjections": {  
    "selectors": [  
        {  
            "targetIndexName": "<YOUR_TARGET_INDEX_NAME_HERE>",  
            "parentKeyFieldName": "ParentKey", // Change this to the name of the field in  
            // your index definition where the parent key will be stored  
            "sourceContext": "/document/pages/*",  
            "mappings": [  
                {  
                    "name": "aml_vector", // Change this to the name of the field in your  
                    // index definition where the Cohere embedding will be stored  
                    "source": "/document/pages/*/aml_vector_data/float/0"  
                }  
            ]  
        },  
        ],  
        "parameters": {}  
    }  
}
```

Sample vectorizer payload

The [Microsoft Foundry model catalog vectorizer](#), unlike the AML skill, is tailored to work only with embedding models that are deployable via the model catalog. The main difference is that you don't have to worry about the request and response payload. However, you must provide the `modelName`, which corresponds to the "Model ID" that you copied after deploying the model.

Here's a sample payload of how you would configure the vectorizer on your index definition given the properties copied from Foundry.

For Cohere models, you should NOT add the `/v1/embed` path to the end of your URL like you did with the skill.

JSON

```
"vectorizers": [  
    {  
        "name": "<YOUR_VECTORIZER_NAME_HERE>",  
        "kind": "aml",  
        "model": "aml_id",  
        "modelType": "cohere",  
        "modelVersion": "1.0",  
        "modelPath": "https://models.foundry.ai/v1/models/aml_id/versions/1.0/embed",  
        "modelFormat": "cohere",  
        "modelConfig": {},  
        "modelMetadata": {}  
    }  
]
```

```
        "amlParameters": {  
            "uri": "<YOUR_URL_HERE>",  
            "key": "<YOUR_PRIMARY_KEY_HERE>",  
            "modelName": "<YOUR_MODEL_ID_HERE>"  
        },  
    },  
]
```

Connect using token authentication

If you can't use key-based authentication, you can configure the AML skill and Microsoft Foundry model catalog vectorizer connection for [token authentication](#) via role-based access control on Azure.

Your search service must have a [system or user-assigned managed identity](#), and the identity must have **Owner** or **Contributor** permissions for your project. You can then remove the `key` field from your skill and vectorizer definition, replacing it with `resourceId`. If your project and search service are in different regions, also provide the `region` field.

JSON

```
"uri": "<YOUR_URL_HERE>,  
"resourceId":  
"subscriptions/<YOUR_SUBSCRIPTION_ID_HERE>/resourceGroups/<YOUR_RESOURCE_GROUP_NAME_<br/>HERE>/providers/Microsoft.MachineLearningServices/workspaces/<YOUR_AML_WORKSPACE_NAME_<br/>HERE>/onlineendpoints/<YOUR_AML_ENDPOINT_NAME_HERE>",<br/>"region": "westus", // Only needed if project is in different region from search<br/>service
```

ⓘ Note

This integration doesn't currently support token authentication for Cohere models. You must use key-based authentication.

Related content

- [Configure a vectorizer in a search index](#)
- [Configure index projections in a skillset](#)
- [AML skill](#)
- [Foundry vectorizer](#)
- [Skill context and input annotation language](#)

Last updated on 11/21/2025

Vector index size and limits

For each vector field, Azure AI Search constructs an internal vector index using the algorithm parameters specified on the field. Because Azure AI Search imposes quotas on vector index size, you should know how to estimate and monitor vector size to ensure you stay under the limits.

Internally, the physical data structures of a search index include:

- Raw content (used for retrieval patterns requiring nontokenized content)
- Inverted indexes (used for searchable text fields)
- Vector indexes (used for searchable vector fields)

This article explains the limits for the internal vector indexes that back each of your vector fields.

Tip

[Vector optimization techniques](#) are generally available. Use capabilities like narrow data types, scalar and binary quantization, and elimination of redundant storage to reduce your vector quota and storage quota consumption.

Key points about quota and vector index size

- Vector index size is measured in bytes.
- The total storage of your service contains all of your vector index files. Azure AI Search maintains different copies of vector index files for different purposes. We offer other options to reduce the [storage overhead of vector indexes](#) by eliminating some of these copies.
- Vector quotas are enforced on the search service as a whole, per partition. If you add partitions, vector quota also increases. Per-partition vector quotas are higher on newer services. For more information, see [Vector index size limits](#).
- Not all algorithms consume vector index size quota. Vector quotas are established based on memory requirements of Approximate Nearest Neighbor (ANN) search. Vector fields created with the Hierarchical Navigable Small World (HNSW) algorithm need to reside in memory during query execution because of the random-access nature of graph-based traversals. Vector fields using the exhaustive K-Nearest Neighbors (KNN) algorithm are loaded into memory dynamically in pages during query execution and thus don't consume vector quota.

Check partition size and quantity

If you aren't sure what your search service limits are, here are two ways to get that information:

- In the Azure portal, on the search service **Overview** page, both the **Properties** tab and **Usage** tab show partition size and storage, and also vector quota and vector index size.
- In the Azure portal, on the **Scale** page, you can review the number and size of partitions.

Your vector limit varies depending on your [service creation date](#).

Check vector index size

A request for vector metrics is a data plane operation. You can use the Azure portal, REST APIs, or Azure SDKs to get vector usage at the service level through service statistics and for individual indexes.

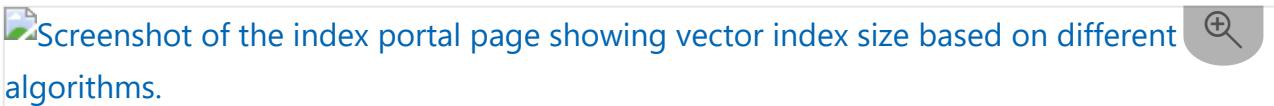


Vector size per index

To get vector index size per index, select **Search management > Indexes** to view a list of indexes and the document count, the size of in-memory vector indexes, and total index size as stored on disk.

Recall that vector quota is based on memory constraints. For vector indexes created using the HNSW algorithm, all searchable vector indexes are permanently loaded into memory. For indexes created using the exhaustive KNN algorithm, vector indexes are loaded in chunks, sequentially, during query time. There's no memory residency requirement for exhaustive KNN indexes. The lifetime of the loaded pages in memory is similar to text search and there are no other metrics applicable to exhaustive KNN indexes other than total storage.

The following screenshot shows two versions of the same vector index. One version is created using HNSW algorithm, where the vector graph is memory resident. Another version is created using exhaustive KNN algorithm. With exhaustive KNN, there's no specialized in-memory vector index, so the portal shows 0 MB for vector index size. Those vectors still exist and are counted in overall storage size, but they don't occupy the in-memory resource that the vector index size metric is tracking.



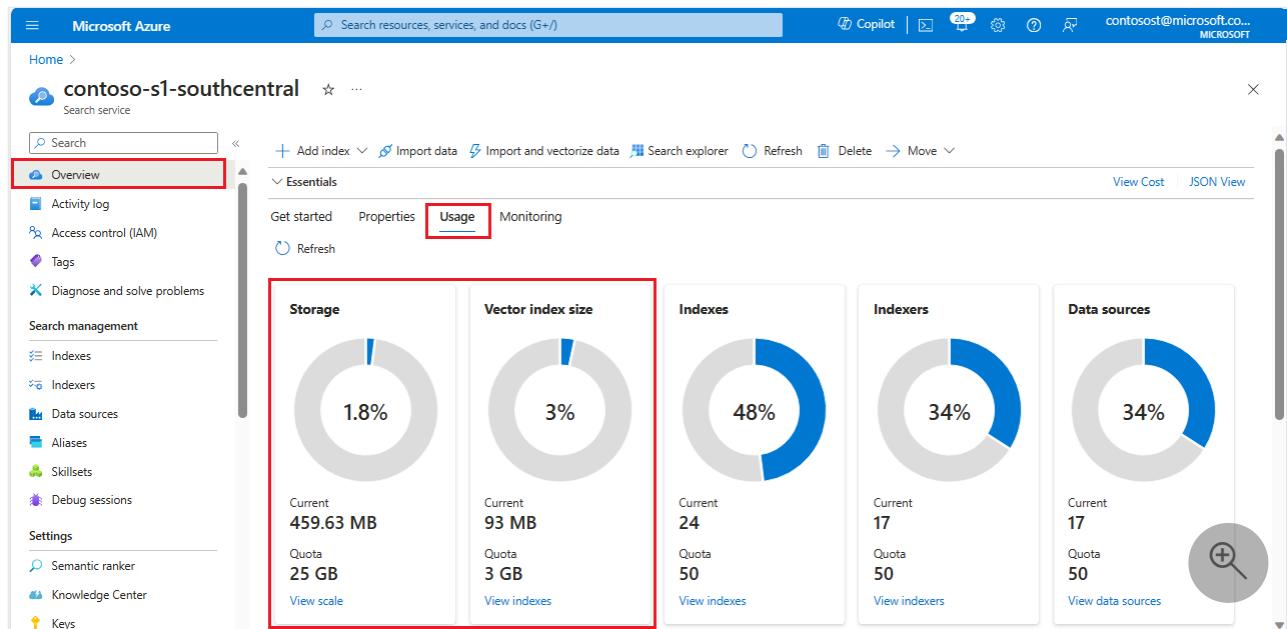
Vector size per service

To get vector index size for the search service as a whole, select the **Overview** page's **Usage** tab. Portal pages refresh every few minutes so if you recently updated an index, wait a bit before checking results.

The following screenshot is for an older Standard 1 (S1) search service, configured for one partition and one replica.

- Storage quota is a disk constraint, and it's inclusive of all indexes (vector and nonvector) on a search service.
- Vector index size quota is a memory constraint. It's the amount of memory required to load all internal vector indexes created for each vector field on a search service.

The screenshot indicates that indexes (vector and nonvector) consume almost 460 megabytes of available disk storage. Vector indexes consume almost 93 megabytes of memory at the service level.



Quotas for both storage and vector index size increase or decrease as you add or remove partitions. If you change the partition count, the tile shows a corresponding change in storage and vector quota.

Note

On disk, vector indexes aren't 93 megabytes. Vector indexes on disk take up about three times more space than vector indexes in memory. See [How vector fields affect disk storage](#) for details.

Factors affecting vector index size

There are three major components that affect the size of your internal vector index:

- Raw size of the data
- Overhead from the selected algorithm
- Overhead from deleting or updating documents within the index

Raw size of the data

Each vector is usually an array of single-precision floating-point numbers, in a field of type `Collection(Edm.Single)`.

Vector data structures require storage, represented in the following calculation as the "raw size" of your data. Use this *raw size* to estimate the vector index size requirements of your vector fields.

The dimensionality of one vector determines its storage size. Multiply the size of one vector by the number of documents containing that vector field to obtain the *raw size*:

```
raw size = (number of documents) * (dimensions of vector field) * (size of data type)
```

[] [Expand table](#)

EDM data type	Size of the data type
<code>Collection(Edm.Single)</code>	4 bytes
<code>Collection(Edm.Half)</code>	2 bytes
<code>Collection(Edm.Int16)</code>	2 bytes
<code>Collection(Edm.SByte)</code>	1 byte

Memory overhead from the selected algorithm

Every ANN algorithm generates extra data structures in memory to enable efficient searching. These structures consume extra space within memory.

For the HNSW algorithm, the memory overhead ranges between 1% and 20% for uncompressed float32 (Edm.Single) vectors.

As dimensionality increases, the memory overhead percentage decreases. This occurs because the raw size of the vectors increases in size while the other data structures, which store graph connectivity information, remain a fixed size for a given m . As a result, the relative impact of these extra data structures diminishes in relation to the overall vector size.

The memory overhead increases with larger values of the HNSW parameter m , which specifies the number of bi-directional links created for each new vector during index construction. This happens because each link contributes approximately 8 to 10 bytes per document, and the total overhead scales proportionally with m .

The following table summarizes the overhead percentages observed in internal tests for *uncompressed* vector fields:

[+] Expand table

Dimensions	HNSW parameter (m)	Overhead percentage
96	4	20%
200	4	8%
768	4	2%
1536	4	1%
3072	4	0.5%

These results demonstrate the relationship between dimensions, HNSW parameter m , and memory overhead for the HNSW algorithm.

For vector fields that use compression techniques, such as [scalar or binary quantization](#), the overhead percentage appears to consume a greater percentage of the total vector index size. As the size of the data decreases, the relative impact of the fixed-size data structures used to store graph connectivity information becomes more significant.

Overhead from deleting or updating documents within the index

When a document with a vector field is either deleted or updated (updates are internally represented as a delete and insert operation), the underlying document is marked as deleted and skipped during subsequent queries. As new documents are indexed and the internal vector

index grows, the system cleans up these deleted documents and reclaims the resources. This means you'll likely observe a lag between deleting documents and the underlying resources being freed.

We refer to this as the *deleted documents ratio*. Since the deleted documents ratio depends on the indexing characteristics of your service, there's no universal heuristic to estimate this parameter, and there's no API or script that returns the ratio in effect for your service. We observe that half of our customers have a deleted documents ratio less than 10%. If you tend to perform high-frequency deletions or updates, then you might observe a higher deleted documents ratio.

This is another factor impacting the size of your vector index. Unfortunately, we don't have a mechanism to surface your current deleted documents ratio.

Estimate total size of data in memory

Taking the previously described factors into account, to estimate the total size of your vector index, use the following calculation:

```
(raw_size) * (1 + algorithm_overhead (in percent)) * (1 + deleted_docs_ratio (in percent))
```

For example, to calculate the `raw_size`, let's assume you're using a popular Azure OpenAI model, `text-embedding-ada-002` with 1,536 dimensions. This means one document would consume 1,536 `Edm.Single` (floats), or 6,144 bytes since each `Edm.Single` is 4 bytes. 1,000 documents with a single, 1,536-dimensional vector field would consume in total 1000 docs x 1536 floats/doc = 1,536,000 floats, or 6,144,000 bytes.

If you have multiple vector fields, you need to perform this calculation for each vector field within your index and add them all together. For example, 1,000 documents with **two** 1,536-dimensional vector fields, consume 1000 docs x **2 fields** x 1536 floats/doc x 4 bytes/float = 12,288,000 bytes.

To obtain the **vector index size**, multiply this `raw_size` by the **algorithm overhead** and **deleted document ratio**. If your algorithm overhead for your chosen HNSW parameters is 10% and your deleted document ratio is 10%, then we get: `6.144 MB * (1 + 0.10) * (1 + 0.10) = 7.434 MB`.

How vector fields affect disk storage

Most of this article provides information about the size of vectors in memory. For information about the storage overhead of vector indexes, see [Eliminate optional vector instances from](#)

storage.

Related content

- [Vector search in Azure AI Search](#)
 - [Choose an approach for optimizing vector storage and processing](#)
-

Last updated on 11/21/2025

Choose an approach for optimizing vector storage and processing

06/12/2025

Embeddings, or the numerical representation of heterogeneous content, are the basis of vector search workloads, but the sizes of embeddings make them hard to scale and expensive to process. Significant research and productization have produced multiple solutions for improving scale and reducing processing times. Azure AI Search taps into a number of these capabilities for faster and cheaper vector workloads.

This article covers all of the optimization techniques in Azure AI Search that can help you reduce vector size and query processing times.

Vector optimization settings are specified in vector field definitions in a search index. Most of the features described in this article are generally available in the [2024-07-01 REST API](#) and Azure SDK packages targeting that version. The [latest preview version](#) adds support for truncated dimensions if you're using text-embedding-3-large or text-embedding-3-small for vectorization.

Evaluate the options

Review the approaches in Azure AI Search for reducing the amount of storage used by vector fields. These approaches aren't mutually exclusive and can be combined for [maximum reduction in vector size](#).

We recommend built-in quantization because it compresses vector size in memory *and* on disk with minimal effort, which tends to provide the most benefit in most scenarios. In contrast, narrow types (except for float16) require special effort to create them, and `stored` saves on disk storage, which isn't as expensive as memory.

 Expand table

Approach	Why use this approach
Add scalar or binary quantization	Compress native float32 or float16 embeddings to int8 (scalar) or byte (binary). This option reduces storage in memory and on disk with no degradation of query performance. Smaller data types, such as int8 or byte, produce vector indexes that are less content-rich than those with larger embeddings. To offset information loss, built-in compression includes options for post-query processing using uncompressed embeddings and oversampling to return more relevant results. Reranking and oversampling are specific features of built-in quantization of float32

Approach	Why use this approach
	or float16 fields and can't be used on embeddings that undergo custom quantization.
Truncate dimensions for MRL-capable text-embedding-3 models (preview)	Use fewer dimensions on text-embedding-3 models. On Azure OpenAI, these models are retrained on the Matryoshka Representation Learning (MRL) technique that produces multiple vector representations at different levels of compression. This approach produces faster searches and reduced storage costs with minimal loss of semantic information. In Azure AI Search, MRL support supplements scalar and binary quantization. When you use either quantization method, you can also specify a <code>truncateDimension</code> property on your vector fields to reduce the dimensionality of text embeddings.
Assign smaller primitive data types to vector fields	Narrow data types, such as float16, int16, int8, and byte (binary), consume less space in memory and on disk, but you must have an embedding model that outputs vectors in a narrow data format. Alternatively, you must have custom quantization logic that outputs small data. A third use case that requires less effort is recasting native float32 embeddings produced by most models to float16. For information about binary vectors, see Index binary vectors .
Eliminate optional storage of retrievable vectors	Vectors returned in a query response are stored separately from vectors used during query execution. If you don't need to return vectors, you can turn off retrievable storage, reducing overall per-field disk storage by up to 50 percent.

All of these options are defined on an empty index. To implement any of them, use the Azure portal, REST APIs, or an Azure SDK package targeting that API version.

After the index is defined, you can load and index documents as a separate step.

Example: Vector size by vector compression technique

[Code sample: Vector quantization and storage options using Python](#) is a Python code sample that creates multiple search indexes that vary by their use of vector storage quantization, [narrow data types](#), and [storage properties](#).

This code creates and compares storage and vector index size for each vector storage optimization option. From these results, you can see that [quantization](#) reduces vector size the most, but the greatest storage savings are achieved if you use multiple options.

[+] [Expand table](#)

Index name	Storage size	Vector size
compressiontest-baseline	21.3613 MB	4.8277 MB

Index name	Storage size	Vector size
compressiontest-scalar-compression	17.7604 MB	1.2242 MB
compressiontest-narrow	16.5567 MB	2.4254 MB
compressiontest-no-stored	10.9224 MB	4.8277 MB
compressiontest-all-options	4.9192 MB	1.2242 MB

Search APIs report storage and vector size at the index level, so indexes and not fields must be the basis of comparison. Use [GET Index Statistics](#) or an equivalent API in the Azure SDKs to obtain vector size.

Related content

- [Get started with REST](#)
- [Supported data types](#)
- [Search REST APIs](#)

Compress vectors using scalar or binary quantization

09/28/2025

Azure AI Search supports scalar and binary quantization for reducing the size of vectors in a search index. Quantization is recommended because it reduces both memory and disk storage for float16 and float32 embeddings. To offset the effects of lossy compression, you can add oversampling and rescore.

To use built-in quantization, follow these steps:

- ✓ Start with [vector fields and a vectorSearch configuration](#) to an index
- ✓ Add `vectorSearch.compressions`
- ✓ Add a `scalarQuantization` or `binaryQuantization` configuration and give it a name
- ✓ Set optional properties to mitigate the effects of lossy indexing
- ✓ Create a new vector profile that uses the named configuration
- ✓ Create a new vector field having the new vector profile
- ✓ Load the index with float32 or float16 data that's quantized during indexing with the configuration you defined
- ✓ Optionally, [query quantized data](#) using the oversampling parameter. If the vector field doesn't specify oversampling in its definition, you can add it at query time.

💡 Tip

[Azure AI Search: Cut Vector Costs Up To 92.5% with New Compression Techniques](#) ↗

compares compression strategies and explains savings in storage and costs. It also includes metrics for measuring relevance based on Normalized discounted cumulative gain (NDCG), demonstrating that you can compress your data without sacrificing search quality.

Prerequisites

- [Vector fields in a search index](#), with a `vectorSearch` configuration specifying either the Hierarchical Navigable Small Worlds (HNSW) or exhaustive K-Nearest Neighbor (KNN) algorithm, and a new vector profile.

Supported quantization techniques

Quantization applies to vector fields receiving float-type vectors. In the examples in this article, the field's data type is `Collection(Edm.Single)` for incoming float32 embeddings, but float16 is also supported. When the vectors are received on a field with compression configured, the engine performs quantization to reduce the footprint of the vector data in memory and on disk.

Two types of quantization are supported:

- Scalar quantization compresses float values into narrower data types. AI Search currently supports int8, which is 8 bits, reducing vector index size fourfold.
- Binary quantization converts floats into binary bits, which takes up 1 bit. This results in up to 28 times reduced vector index size.

 **Note**

While free services support quantization, they don't demonstrate the full storage savings due to the limited storage quota.

How scalar quantization works in Azure AI Search

Scalar quantization reduces the resolution of each number within each vector embedding. Instead of describing each number as a 16-bit or 32-bit floating point number, it uses an 8-bit integer. It identifies a range of numbers (typically 99th percentile minimum and maximum) and divides them into a finite number of levels or bin, assigning each bin an identifier. In 8-bit scalar quantization, there are 2^8 , or 256, possible bins.

Each component of the vector is mapped to the closest representative value within this set of quantization levels in a process akin to rounding a real number to the nearest integer. In the quantized 8-bit vector, the identifier number stands in place of the original value. After quantization, each vector is represented by an array of identifiers for the bins to which its components belong. These quantized vectors require much fewer bits to store compared to the original vector, thus reducing storage requirements and memory footprint.

How binary quantization works in Azure AI Search

Binary quantization compresses high-dimensional vectors by representing each component as a single bit, either 0 or 1. This method drastically reduces the memory footprint and accelerates vector comparison operations, which are crucial for search and retrieval tasks. Benchmark tests show up to 96% reduction in vector index size.

It's particularly effective for embeddings with dimensions greater than 1024. For smaller dimensions, we recommend testing the quality of binary quantization, or trying scalar instead. Additionally, we've found binary quantization performs very well when embeddings are centered around zero. Most popular embedding models offered by OpenAI, Cohere, and Mistral are centered around zero.

Supported rescoring techniques

Rescoring is an optional technique used to offset information loss due to vector quantization. During query execution, it uses oversampling to pick up extra vectors, and supplemental information to rescore initial results found by the query. Supplemental information is either uncompressed original full-precision vectors - or for binary quantization only - you have the option of rescoring using the binary quantized document candidates against the query vector.

Only HNSW graphs allow rescoring. Exhaustive KNN doesn't support rescoring because by definition, all vectors are scanned at query time, which makes rescoring and oversampling irrelevant.

Rescoring options are specified in the index, but you can invoke rescoring at query time by adding the oversampling query parameter.

[] Expand table

Object	Properties
Index	Add RescoringOptions to the vector compressions section. The examples in this article use <code>RescoringOptions</code> .
Query	Add <code>oversampling</code> on RawVectorQuery or VectorizableTextQuery definitions. Adding <code>oversampling</code> invokes rescoring at query time.

! Note

Rescoring parameter names have changed over the last several releases. If you're using an older preview API, review the [upgrade instructions](#) for addressing breaking changes.

The generalized process for rescoring is:

1. The vector query executes over compressed vector fields.
2. The vector query returns the top k oversampled candidates.
3. Oversampled k candidates are rescored using either the uncompressed original vectors for scalar quantization, or the dot product of binary quantization.

4. After rescoring, results are adjusted so that more relevant matches appear first.

Oversampling for scalar quantized vectors requires the availability of the original full precision vectors. Oversampling for binary quantized vectors can use either full precision vectors (`preserveOriginals`) or the dot product of the binary vector (`discardOriginals`). If you're optimizing vector storage, make sure to keep the full precision vectors in the index if you need them for rescoring purposes. For more information, see [Eliminate optional vector instances from storage](#).

Add "compressions" to a search index

This section explains how to specify a `vectorsSearch.compressions` section in the index. The following example shows a partial index definition with a `fields` collection that includes a vector field.

The compression example includes both `scalarQuantization` or `binaryQuantization`. You can specify as many compression configurations as you need, and then assign the ones you want to a vector profile.

Syntax for `vectorSearch.Compressions` varies between stable and preview REST APIs, with the preview adding more options for storage optimization, plus changes to existing syntax. Backwards compatibility is preserved through internal API mappings, but we recommend adopting the newer properties in code that targets 2024-11-01-preview and future versions.

Use the [Create Index](#) or [Create or Update Index](#) REST API to configure compression settings.

HTTP

```
POST https://[servicename].search.windows.net/indexes?api-version=2025-09-01

{
    "name": "my-index",
    "description": "This is a description of this index",
    "fields": [
        { "name": "Id", "type": "Edm.String", "key": true, "retrievable": true,
"searchable": true, "filterable": true },
        { "name": "content", "type": "Edm.String", "retrievable": true, "searchable": true },
        { "name": "vectorContent", "type": "Collection(Edm.Single)", "retrievable": false, "searchable": true, "dimensions": 1536, "vectorSearchProfile": "vector-profile-1" },
    ],
    "vectorSearch": {
        "profiles": [
            {
                "name": "vector-profile-1",
                "algorithm": "use-hnsw",
            }
        ]
    }
}
```

```

        "compression": "use-scalar"
    }
],
"algorithms": [
{
    "name": "use-hnsw",
    "kind": "hnsw",
    "hnswParameters": { },
    "exhaustiveKnnParameters": null
}
],
"compressions": [
{
    "scalarQuantizationParameters": {
        "quantizedDataType": "int8"
    },
    "name": "mySQ8",
    "kind": "scalarQuantization",
    "rescoringOptions": {
        "enableRescoring": true,
        "defaultOversampling": 10,
        "rescoreStorageMethod": "preserveOriginals"
    },
    "truncationDimension": 2
},
{
    "name": "myBQC",
    "kind": "binaryQuantization",
    "rescoringOptions": {
        "enableRescoring": true,
        "defaultOversampling": 10,
        "rescoreStorageMethod": "discardOriginals"
    },
    "truncationDimension": 2
}
]
},
}

```

Key points:

- `kind` must be set to `scalarQuantization` or `binaryQuantization`.
- `rescoringOptions` are a collection of properties used to offset lossy compression by rescoring query results using the original full-precision vectors that exist prior to quantization. For rescoring to work, you must have the vector instance that provides this content. Setting `rescoreStorageMethod` to `discardOriginals` prevents you from using `enableRescoring` or `defaultOversampling`. For more information about vector storage, see [Eliminate optional vector instances from storage](#).

- `"rescoreStorageMethod": "preserveOriginals"` rescores vector search results with the original full-precision vectors can result in adjustments to search score and rankings, promoting the more relevant matches as determined by the rescoring step. For binary quantization, you can set `rescoreStorageMethod` to `discardOriginals` to further reduce storage, without reducing quality. Original vectors aren't needed for binary quantization.
- `defaultOversampling` considers a broader set of potential results to offset the reduction in information from quantization. The formula for potential results consists of the `k` in the query, with an oversampling multiplier. For example, if the query specifies a `k` of 5, and oversampling is 20, then the query effectively requests 100 documents for use in reranking, using the original uncompressed vector for that purpose. Only the top `k` reranked results are returned. This property is optional. Default is 4.
- `quantizedDataType` is optional and applies to scalar quantization only. If you add it, it must be set to `int8`. This is the only primitive data type supported for scalar quantization at this time. Default is `int8`.
- `truncationDimension` taps inherent capabilities of the text-embedding-3 models to "encode information at different granularities and allows a single embedding to adapt to the computational constraints of downstream tasks" (see [Matryoshka Representation Learning](#)). You can use truncated dimensions with or without rescoring options. For more information about how this feature is implemented in Azure AI Search, see [Truncate dimensions using MRL compression](#).

Add the vector search algorithm

You can use the HNSW or eKNN algorithm in the 2024-11-01-preview REST API or later. For the stable version, use HNSW only. If you want rescoring, you must choose HNSW.

JSON

```
"vectorSearch": {
  "profiles": [ ],
  "algorithms": [
    {
      "name": "use-hnsw",
      "kind": "hnsw",
      "hnswParameters": {
        "m": 4,
        "efConstruction": 400,
        "efSearch": 500,
        "metric": "cosine"
      }
    }
  ],
}
```

```
        "compressions": [ <see previous section> ]  
    }
```

Create and assign a new vector profile

To use a new quantization configuration, you must create a *new* vector profile. Creation of a new vector profile is necessary for building compressed indexes in memory. Your new profile uses HNSW.

1. In the same index definition, create a new vector profile and add a compression property and an algorithm. Here are two profiles, one for each quantization approach.

JSON

```
"vectorSearch": {  
    "profiles": [  
        {  
            "name": "vector-profile-hnsw-scalar",  
            "compression": "use-scalar",  
            "algorithm": "use-hnsw",  
            "vectorizer": null  
        },  
        {  
            "name": "vector-profile-hnsw-binary",  
            "compression": "use-binary",  
            "algorithm": "use-hnsw",  
            "vectorizer": null  
        }  
    ],  
    "algorithms": [ <see previous section> ],  
    "compressions": [ <see previous section> ]  
}
```

2. Assign a vector profile to a *new* vector field. The data type of the field is either float32 or float16.

In Azure AI Search, the Entity Data Model (EDM) equivalents of float32 and float16 types are `Collection(Edm.Single)` and `Collection(Edm.Half)`, respectively.

JSON

```
{  
    "name": "vectorContent",  
    "type": "Collection(Edm.Single)",  
    "searchable": true,  
    "retrievable": true,  

```

```
        "vectorSearchProfile": "vector-profile-hnsw-scalar",  
    }
```

3. Load the index using indexers for pull model indexing, or APIs for push model indexing.

Query a quantized vector field using oversampling

Query syntax for a compressed or quantized vector field is the same as for noncompressed vector fields, unless you want to override parameters associated with oversampling and rescoring. You can add an `oversampling` parameter to invoke oversampling and rescoring at query time.

Use [Search Documents](#) for the request.

Recall that the [vector compression definition](#) in the index has settings for `enableRescoring`, `rescoreStorageMethod`, and `defaultOversampling` to mitigate the effects of lossy compression. You can override the default values to vary the behavior at query time. For example, if `defaultOversampling` is 10.0, you can change it to something else in the query request.

You can set the oversampling parameter even if the index doesn't explicitly have rescoring options or `defaultOversampling` definition. Providing `oversampling` at query time overrides the index settings for that query and executes the query with an effective `enableRescoring` as true.

HTTP

```
POST https://[service-name].search.windows.net/indexes/demo-index/docs/search?api-version=2025-09-01  
  
{  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [8, 2, 3, 4, 3, 5, 2, 1],  
            "fields": "myvector",  
            "oversampling": 12.0,  
            "k": 5  
        }  
    ]  
}
```

Key points:

- Oversampling applies to vector fields that undergo vector compression, per the vector profile assignment.

- Oversampling in the query overrides the `defaultOversampling` value in the index, or invokes oversampling and rescoring at query time, even if the index's compression configuration didn't specify oversampling or reranking options.

Index binary vectors for vector search

09/29/2025

Azure AI Search supports a packed binary type of `Collection(Edm.Byte)` for further reducing the storage and memory footprint of vector data. You can use this data type for output from models such as [Cohere's Embed v3 binary embedding models](#) or any other embedding model or process that outputs vectors as binary bytes.

There are three steps to configuring an index for binary vectors:

- ✓ Add a vector search algorithm that specifies Hamming distance for binary vector comparison
- ✓ Add a vector profile that points to the algorithm
- ✓ Add a vector field of type `Collection(Edm.Byte)` and assign the Hamming distance

This article assumes you're familiar with [creating an index in Azure AI Search](#) and [adding vector fields](#). It uses the REST APIs to illustrate each step, but you could also add a binary field to an index in the Azure portal or Azure SDK.

The binary data type is assigned to fields using the [Create Index](#) or [Create Or Update Index](#) APIs.

Tip

If you're investigating binary vector support for its smaller footprint, you might also consider the vector quantization and storage reduction features in Azure AI Search. Inputs are float32 or float16 embeddings. Output is stored data in a much smaller format. For more information, see [Compress using binary or scalar quantization](#) and [Assign narrow data types](#).

Prerequisites

- Binary vectors, with 1 bit per dimension, packaged in uint8 values with 8 bits per value. These can be obtained by using models that directly generate *packaged binary* vectors, or by quantizing vectors into binary vectors client-side during indexing and searching.

Limitations

- No Azure portal support in the **Import data (new)** wizard.

- No support for binary fields in the [AML skill](#) that's used for integrated vectorization of models in the Azure AI Foundry model catalog.

Add a vector search algorithm and vector profile

Vector search algorithms are used to create the query navigation structures during indexing. For binary vector fields, vector comparisons are performed using the Hamming distance metric.

1. To add a binary field to an index, set up a [Create or Update Index](#) request using the REST API or the Azure portal.
2. In the index schema, add a `vectorSearch` section that specifies profiles and algorithms.
3. Add one or more [vector search algorithms](#) that have a similarity metric of `hamming`. It's common to use Hierarchical Navigable Small Worlds (HNSW), but you can also use Hamming distance with exhaustive K-Nearest Neighbors (KNN).
4. Add one or more vector profiles that specify the algorithm.

The following example shows a basic `vectorSearch` configuration:

JSON

```
"vectorSearch": {  
    "profiles": [  
        {  
            "name": "myHnswProfile",  
            "algorithm": "myHnsw",  
            "compression": null,  
            "vectorizer": null  
        }  
    ],  
    "algorithms": [  
        {  
            "name": "myHnsw",  
            "kind": "hnsw",  
            "hnswParameters": {  
                "metric": "hamming"  
            }  
        },  
        {  
            "name": "myExhaustiveKnn",  
            "kind": "exhaustiveKnn",  
            "exhaustiveKnnParameters": {  
                "metric": "hamming"  
            }  
        }  
    ]  
}
```

```
    ]  
}
```

Add a binary field to an index

The fields collection of an index must include a field for the document key, vector fields, and any other fields that you need for hybrid search scenarios.

Binary fields are of type `Collection(Edm.Byte)` and contain embeddings in packed form. For example, if the original embedding dimension is `1024`, the packed binary vector length is `ceiling(1024 / 8) = 128`. You get the packed form by setting the `vectorEncoding` property on the field.

- ✓ Add a field to the fields collection and give it name.
- ✓ Set data type to `Collection(Edm.Byte)`.
- ✓ Set `vectorEncoding` to `packedBit` for binary encoding.
- ✓ Set `dimensions` to `1024`. Specify the original (unpacked) vector dimension.
- ✓ Set `vectorSearchProfile` to a profile you defined in the previous step.
- ✓ Make the field searchable.

The following field definition is an example of the properties you should set:

JSON

```
"fields": [  
    . . .  
    {  
        "name": "my-binary-vector-field",  
        "type": "Collection(Edm.Byte)",  
        "vectorEncoding": "packedBit",  
        "dimensions": 1024,  
        "vectorSearchProfile": "myHnswProfile",  
        "searchable": true  
    },  
    . . .  
]
```

See also

Code samples in the [azure-search-vector-samples](#) repository demonstrate end-to-end workflows that include schema definition, vectorization, indexing, and queries.

There's demo code for [Python](#), [C#](#), and [JavaScript](#).

Assign narrow data types to vector fields in Azure AI Search

06/12/2025

An easy way to reduce vector size is to store embeddings in a smaller data format. Most embedding models output 32-bit floating point numbers. However, if you quantize your vectors or use an embedding model that natively supports quantization, the output might be float16, int16, or int8, which are significantly smaller than float32. You can accommodate these smaller vector sizes by assigning a narrow data type to a vector field. In the vector index, narrow data types consume less storage.

Data types are assigned to fields in an index definition. You can use the Azure portal, the [Search REST APIs](#), or an Azure SDK package that provides the feature.

Prerequisites

- An embedding model that outputs small data formats, such as text-embedding-3 or Cohere V3 embedding models.

Supported narrow data types

1. Review the [data types used for vector fields](#) for recommended usage:

- `Collection(Edm.Single)` 32-bit floating point (default)
- `Collection(Edm.Half)` 16-bit floating point (narrow)
- `Collection(Edm.Int16)` 16-bit signed integer (narrow)
- `Collection(Edm.SByte)` 8-bit signed integer (narrow)
- `Collection(Edm.Byte)` 8-bit unsigned integer (only allowed with packed binary data types)

2. From that list, determine which data type is valid for your embedding model's output or for vectors that undergo custom quantization.

The following table provides links to several embedding models that can use a narrow data type (`Collection(Edm.Half)`) without extra quantization. You can cast from float32 to float16 (using `Collection(Edm.Half)`) with no extra work.

[] Expand table

Embedding model	Native output	Assign this type in Azure AI Search
text-embedding-ada-002	Float32	Collection(Edm.Single) or Collection(Edm.Half)
text-embedding-3-small	Float32	Collection(Edm.Single) or Collection(Edm.Half)
text-embedding-3-large	Float32	Collection(Edm.Single) or Collection(Edm.Half)
Cohere V3 embedding models with int8 embedding_type ↴	Int8	Collection(Edm.SByte)

You can use other narrow data types if your model emits embeddings in the smaller data format or if you have custom quantization that converts vectors to a smaller format.

3. Make sure you understand the tradeoffs of a narrow data type. `Collection(Edm.Half)` has less information, which results in lower resolution. If your data is homogeneous or dense, losing extra detail or nuance could lead to unacceptable results at query time because there's less detail that can be used to distinguish nearby vectors apart.

Assign the data type

[Define and build an index](#). You can use the Azure portal, [Create or Update Index \(REST API\)](#), or an Azure SDK package for this step.

This field definition uses a narrow data type, `Collection(Edm.Half)`, that can accept a float32 embedding stored as a float16 value. As is true for all vector fields, `dimensions` and `vectorSearchProfile` are set. The specifics of the `vectorSearchProfile` are immaterial to the datatype.

We recommend that you set `retrievable` and `stored` to true if you want to visually check the values of the field. On a subsequent rebuild, you can change these properties to false for reduced storage requirements.

JSON

```
{
  "name": "nameEmbedding",
  "type": "Collection(Edm.Half)",
  "searchable": true,
  "filterable": false,
  "retrievable": true,
  "sortable": false,
```

```
"facetable": false,  
"key": false,  
"indexAnalyzer": null,  
"searchAnalyzer": null,  
"analyzer": null,  
"synonymMaps": [],  
"dimensions": 1536,  
"vectorSearchProfile": "myHnswProfile"  
}
```

Recall that vector fields aren't filterable, sortable, or facetable. They can't be used as keys and don't use analyzers or synonym maps.

Working with a production index

Data types are assigned on new fields when they're created. You can't change the data type of an existing field, and you can't drop a field without [rebuilding the index](#). For established indexes already in production, it's common to work around this issue by creating new fields with the desired revisions and then removing obsolete fields during a planned index rebuild.

Check results

1. Verify the field content matches the data type. Assuming the vector field is marked as [retrievable](#), use [Search explorer](#) or [Search - POST](#) to return vector field content.
2. To check vector index size, refer to the vector index size column on the [Search management > Indexes](#) page in the [Azure portal](#). Alternatively, you can use [GET Index Statistics \(REST API\)](#) or an equivalent Azure SDK method.

! Note

The field's data type is used to create the physical data structure. If you want to change a data type later, either [drop and rebuild the index](#) or create a second field with the new definition.

Eliminate optional vector instances from storage

09/29/2025

Azure AI Search stores multiple copies of vector fields that are used in specific workloads. If your search scenarios don't require all of these copies, you can omit storage for that workload.

Use cases where an extra copy is used include:

- Returning raw vectors in a query response or supporting incremental updates to vector content.
- Rescoring compressed (quantized) vectors as a query optimization technique.

Removing storage is irreversible and requires reindexing if you want it back.

Prerequisites

- [Vector fields in a search index](#), with a `vectorSearch` configuration specifying either the Hierarchical Navigable Small Worlds (HNSW) or exhaustive K-Nearest Neighbor (KNN) algorithm, and a new vector profile.

How vector fields are stored

[Expand table](#)

Instance	Usage	Required for search	How removed
Vectors in the HNSW graph for Approximate Nearest Neighbors (ANN) search (HNSW graph) or vectors for exhaustive K-Nearest Neighbors (eKNN index)	Used for query execution. Consists of either full-precision vectors (when no compression is applied) or quantized vectors.	Essential	There are no parameters for removing this instance.

Instance	Usage	Required for search	How removed
Source vectors received during document indexing (JSON data)	Used for incremental data refresh with <code>merge</code> or <code>mergeOrUpload</code> indexing actions. Also used to return "retrievable" vectors in the query response.	No	Set <code>stored</code> property to false.
Original full-precision vectors (binary data) ¹	For compressed vectors, it's used for <code>preserveOriginals</code> rescore on an oversampled candidate set of results from ANN search. This applies to vector fields that undergo scalar or binary quantization , and it applies to queries using the HNSW graph. If you're using eKNN, all vectors are in scope for the query, so rescore has no effect and thus not supported.	No	Set <code>rescoringOptions.rescoreStorageMethod</code> property to <code>discardOriginals</code> in <code>vectorSearch.compressions</code> .

¹ This copy is also for internal index operations and for exhaustive KNN search in older API versions, on indexes created using the 2023 APIs. On newer indexes, an eKNN-configured field consists of full-precision vectors so no extra copy is needed.

Remove source vectors (JSON data)

In a vector field definition, `stored` is a boolean property that determines whether storage is allocated for retrievable vector content obtained during indexing (the source instance). By default, `stored` is set to `true`. If you don't need raw vector content in a query response, changing `stored` to `false` can save up to 50% storage per field.

Considerations for setting `"stored": false`:

- Because vectors aren't human readable, you can generally omit them from results sent to LLMs in RAG scenarios or from results rendered on a search page. However, you should keep them if you're using vectors in a downstream process that consumes vector content.
- If your indexing strategy uses [partial document updates](#), such as `merge` or `mergeOrUpload` on an existing document, setting `"stored": false` prevents content updates to those

fields during the merge. You must include the entire vector field (and nonvector fields you're updating) in each reindexing operation. Otherwise, the vector data is lost without an error or warning. To avoid this risk altogether, set "stored": true.

ⓘ Important

Setting the "stored": false attribution is irreversible. This property can only be set when you create the index and is only allowed on vector fields. Updating an existing index with new vector fields can't set this property to false. If you want retrievable vector content later, you must drop and rebuild the index or create and load a new field that has the new attribution.

For new vector fields in a search index, set "stored": false to permanently remove retrievable storage for the vector field. The following example shows a vector field definition with the stored property.

HTTP

```
PUT https://[service-name].search.windows.net/indexes/demo-index?api-version=2025-09-01@search.rerankerBoostedScore
Content-Type: application/json
api-key: [admin key]

{
  "name": "demo-index",
  "fields": [
    {
      "name": "vectorContent",
      "type": "Collection(Edm.Single)",
      "retrievable": false,
      "stored": false,
      "dimensions": 1536,
      "vectorSearchProfile": "vectorProfile"
    }
  ]
}
```

Summary of key points

- Applies to fields that have a [vector data type](#).
- Affects storage on disk, not memory, and has no effect on queries. Query execution uses a separate vector index that's unaffected by the stored property because that copy of the vector is always stored.

- The `stored` property is set during index creation on vector fields and is irreversible. If you want retrievable content later, you must drop and rebuild the index or create and load a new field that has the new attribution.
- Defaults are `"stored": true` and `" retrievable": false`. In a default configuration, a retrievable copy is stored but isn't automatically returned in results. When `stored` is `true`, you can toggle `retrievable` between `true` and `false` at any time without having to rebuild an index. When `stored` is `false`, `retrievable` must be `false` and can't be changed.

Remove full-precision vectors

Original full-precision vectors are used in rescoring operations over compressed (quantized) vectors. The intent of rescoring is to mitigate the loss in information due to compression. The effect of rescoring is retrieval of a larger set of candidate documents from the compressed index, with recomputation of similarity scores using the original vectors or the dot product. For rescoring to work, original vectors must be retained in storage for certain scenarios. As a result, while quantization reduces memory usage (vector index size usage), it slightly increases storage requirements since both compressed and original vectors are stored. The extra storage is approximately equal to the size of the compressed index.

Rescoring requirements by quantization approach:

- Rescoring of scalar quantized vectors requires retention of the original full-precision vectors.
- Rescoring of binary quantized vectors can use either the original full-precision vectors, or the dot product of the binary embedding, which produces high quality search results, without having to reference full-precision vectors in the index.

Rescoring recommendations:

- For scalar quantization, preserve original full-precision vectors in the index because they're required for rescore.
- For binary quantization, either preserve original full-precision vectors for the highest quality of rescoring, or discard full-precision vectors if you want to rescore based on the dot product of the binary embeddings.

The `rescoreStorageMethod` property controls whether full-precision vectors are stored. In `vectorSearch.compressions`, the `rescoreStorageMethod` property is set to `preserveOriginals` by default, which retains full-precision vectors for [oversampling and rescoring capabilities](#) to reduce the effect of lossy compression on the HNSW graph. If you don't need rescoring, or if

you used binary quantization and the dot product for rescoring, you can reduce vector storage by setting `rescoreStorageMethod` to `discardOriginals`.

Important

Setting the `rescoreStorageMethod` property is irreversible and can adversely affect search quality, although the degree depends on the compression method and any mitigations you apply.

To set this property:

1. Use [Create Index](#) or [Create or Update Index](#) REST APIs, or an Azure SDK.
2. Add a `vectorSearch` section to your index with profiles, algorithms, and compressions.
3. Under `vectorSearch.compressions`, add `rescoringOptions` with `enableRescoring` set to true, `defaultOversampling` set to a positive integer, and `rescoreStorageMethod` set to `discardOriginals` for binary quantization and `preserveOriginals` for scalar quantization.

HTTP

```
PUT https://[service-name].search.windows.net/indexes/demo-index?api-version=2025-09-01
```

```
{
  "name": "demo-index",
  "fields": [ . . . ],
  . .
  "vectorSearch": {
    "profiles": [
      {
        "name": "myVectorProfile-1",
        "algorithm": "myHnsw",
        "compression": "myScalarQuantization"
      },
      {
        "name": "myVectorProfile-2",
        "algorithm": "myHnsw",
        "compression": "myBinaryQuantization"
      }
    ],
    "algorithms": [
      {
        "name": "myHnsw",
        "kind": "hnsw",
        "hnswParameters": {
          "metric": "cosine",
          "m": 4,
          "efConstruction": 400,
        }
      }
    ]
  }
}
```

```
        "efSearch": 500
    },
    "exhaustiveKnnParameters": null
}
],
"compressions": [
{
    "name": "myScalarQuantization",
    "kind": "scalarQuantization",
    "rescoringOptions": {
        "enableRescoring": true,
        "defaultOversampling": 10,
        "rescoreStorageMethod": "preserveOriginals"
    },
    "scalarQuantizationParameters": {
        "quantizedDataType": "int8"
    },
    "truncationDimension": null
},
{
    "name": "myBinaryQuantization",
    "kind": "binaryQuantization",
    "rescoringOptions": {
        "enableRescoring": true,
        "defaultOversampling": 10,
        "rescoreStorageMethod": "discardOriginals"
    },
    "truncationDimension": null
}
]
}
```

ⓘ Note

Vector storage strategies have been evolving over the last several releases. Index creation date and API version determine your storage options. For example, in the 2024-11-01-preview, if you set `discardOriginals` to remove full-precision vectors, there was no rescoring for binary quantization because the dot product approach wasn't available. We recommend using the latest APIs for the best mitigation options.

Truncate dimensions using MRL compression

10/08/2025

Exercise the ability to use fewer dimensions on text-embedding-3 models. On Azure OpenAI, text-embedding-3 models are retrained on the [Matryoshka Representation Learning](#) (MRL) technique that produces multiple vector representations at different levels of compression. This approach produces faster searches and reduced storage costs with minimal loss of semantic information.

In Azure AI Search, MRL support supplements [scalar and binary quantization](#). When you use either quantization method, you can specify a `truncationDimension` property on your vector fields to reduce the dimensionality of text embeddings.

MRL multilevel compression saves on vector storage and improves query response times for vector queries based on text embeddings. In Azure AI Search, MRL support is only offered together with another method of quantization. Using binary quantization with MRL provides the maximum vector index size reduction. To achieve maximum storage reduction, use binary quantization with MRL and set `stored` to `false`.

⚠ Warning

If you set `stored` to `false`, vector data is lost during partial document updates unless you provide the entire vector in each update. Set `stored` to `true` to avoid this issue. For more information, see [Eliminate optional vector instances from storage](#).

Prerequisites

- A text-embedding-3 model, such as text-embedding-3-small or text-embedding-3-large.
- A [supported client](#).
- [New vector fields](#) of type `Edm.Half` or `Edm.Single`. You can't add MRL compression to an existing field.
- [Hierarchical Navigable Small World \(HNSW\) algorithm](#).
- [Scalar or binary quantization](#). Truncated dimensions can be set only when scalar or binary quantization is configured. We recommend binary quantization for MRL compression.

Supported clients

You can use the REST APIs or Azure SDK packages to implement MRL compression. At this time, there's no Azure portal or Azure AI Foundry support.

- Check the change logs for each Azure SDK package for feature support: [Python](#), [.NET](#), [Java](#), [JavaScript](#).

Use MRL-extended text embeddings

MRL is built into the text embedding model you're already using. To use MRL capabilities in Azure AI Search:

1. Use [Create or Update Index](#) or an equivalent API to specify the index schema.
2. [Add vector fields](#) to the index definition.
3. Specify a `vectorSearch.compressions` object in your index definition.
4. Include a quantization method, either scalar or binary (recommended).
5. Include the `truncationDimension` parameter and set it to 512. If you're using the text-embedding-3-large model, you can set it as low as 256.
6. Include a vector profile that specifies the HNSW algorithm and the vector compression object.
7. Assign the vector profile to a vector field of type `Edm.Half` or `Edm.Single` in the fields collection.

There are no query-side modifications for using an MRL-capable text embedding model. MRL support doesn't affect integrated vectorization, text-to-query conversions at query time, semantic ranking, and other relevance-enhancement features, such as reranking with original vectors and oversampling.

Although indexing is slower due to the extra steps, queries are faster.

Example: Vector search configuration that supports MRL

The following example illustrates a vector search configuration that meets the requirements and recommendations of MRL.

`truncationDimension` is a compression property. It specifies how much to shrink the vector graph in memory together with a compression method like scalar or binary compression. We recommend 1,024 or higher for `truncationDimension` with binary quantization. A dimensionality of less than 1,000 degrades the quality of search results when using MRL and binary compression.

JSON

```
{
  "vectorSearch": {
    "profiles": [
      {
        "name": "use-bq-with-mrl",
        "compression": "use-mrl,use-bq",
        "algorithm": "use-hnsw"
      }
    ],
    "algorithms": [
      {
        "name": "use-hnsw",
        "kind": "hnsw",
        "hnswParameters": {
          "m": 4,
          "efConstruction": 400,
          "efSearch": 500,
          "metric": "cosine"
        }
      }
    ],
    "compressions": [
      {
        "name": "use-mrl",
        "kind": "binaryQuantization",
        "rescoringOptions": {
          "enableRescoring": true,
          "defaultOversampling": 10,
          "rescoreStorageMethod": "preserveOriginals"
        },
        "truncationDimension": 1024
      },
      {
        "name": "use-bq",
        "kind": "binaryQuantization",
        "rescoringOptions": {
          "enableRescoring": true,
          "defaultOversampling": 10,
          "rescoreStorageMethod": "discardOriginals"
        }
      }
    ]
  }
}
```

```
    }  
}
```

Here's an example of a [fully specified vector field definition](#) that satisfies the requirements for MRL. Recall that vector fields must:

- Be of type `Edm.Half` or `Edm.Single`.
- Have a `vectorSearchProfile` property that specifies the algorithm and compression settings.
- Have a `dimensions` property that specifies the number of dimensions for scoring and ranking results. Its value should be the dimensions limit of the model you're using (1,536 for text-embedding-3-small).

JSON

```
{  
  "name": "text_vector",  
  "type": "Collection(Edm.Single)",  
  "searchable": true,  
  "filterable": false,  
  "retrievable": false,  
  "stored": false,  
  "sortable": false,  
  "facetable": false,  
  "key": false,  
  "indexAnalyzer": null,  
  "searchAnalyzer": null,  
  "analyzer": null,  
  "normalizer": null,  
  "dimensions": 1536,  
  "vectorSearchProfile": "use-bq-with-mrl",  
  "vectorEncoding": null,  
  "synonymMaps": []  
}
```

Import data wizards in the Azure portal

i Important

We're consolidating the Azure AI Search wizards. Key changes include:

- The **Import and vectorize data** wizard is now called **Import data (new)**.
- The **Import data** workflow is now available in **Import data (new)**.

The **Import data** wizard will eventually be deprecated. For now, you can still use this wizard, but we recommend the new wizard for an improved search experience that uses the latest frameworks.

The wizards don't have identical keyword search workflows. Certain skills and capabilities are only available in the old wizard. For more information about their similarities and differences, continue reading this article.

Azure AI Search has two wizards that automate indexing, enrichment, and object creation for various search scenarios:

- The **Import data** wizard supports keyword (nonvector) search. You can extract text and numbers from raw documents. You can also configure applied AI and built-in skills to infer structure and generate searchable text from image files and unstructured data.
- The **Import data (new)** wizard supports keyword search, RAG, and multimodal RAG. For keyword search, it modernizes the **Import data** workflow but lacks some functionality, such as automatic metadata field creation. For RAG and multimodal RAG, it connects to your embedding model deployment, sends requests, and generates vectors from text or images.

Despite their differences, the wizards follow similar workflows for content ingestion and indexing. The following table summarizes their capabilities.

[] Expand table

Capability	Import data wizard	Import data (new) wizard
Index creation	✓	✓
Indexer pipeline creation	✓	✓
Azure Logic Apps connectors	✗	✓
Built-in sample data	✗	✗

Capability	Import data wizard	Import data (new) wizard
Skills-based enrichment	✓	✓
Vector and multimodal support	✗	✓
Semantic ranking support	✗	✓
Knowledge store support	✓	✗

Built-in sample data for the hotels sample index is no longer provided, but you can create an identical index by following the [Quickstart: Create an index for keyword search](#).

This article explains how the wizards work to help you with proof-of-concept testing. For step-by-step instructions, see [Try the wizards](#).

Supported data sources and scenarios

This section describes the available options in each wizard.

Data sources

The wizards support the following data sources, most of which use [built-in indexers](#). Exceptions are noted in the table's footnotes.

[] Expand table

Data source	Import data wizard	Import data (new) wizard
ADLS Gen2	✓	✓
Azure Blob Storage	✓	✓
Azure File Storage	✗	✓ ^{1, 2}
Azure Queues	✗	✓ ¹
Azure Table Storage	✓	✓
Azure SQL Database and Managed Instance	✓	✓
Cosmos DB for NoSQL	✓	✓
Cosmos DB for MongoDB	✓	✓
Cosmos DB for Apache Gremlin	✓	✓
MySQL	✗	✗

Data source	Import data wizard	Import data (new) wizard
OneDrive	✗	✓ ¹
OneDrive for Business	✗	✓ ¹
OneLake	✓	✓
Service Bus	✗	✓ ¹
SharePoint	✗	✓ ^{1, 2}
SQL Server on virtual machines	✓	✓

¹ This data source uses an [Azure Logic Apps connector \(preview\)](#) instead of a built-in indexer.

² Instead of using a Logic Apps connector, you can use the Search Service REST APIs to programmatically index data from [Azure File Storage](#) or [SharePoint](#).

Skills

Each wizard generates a skillset and outputs field mappings based on options you select. After the skillset is created, you can modify its JSON definition to add or remove skills.

The following skills might appear in a wizard-generated skillset.

[Expand table](#)

Skill	Import data wizard	Import data (new) wizard
Azure Vision multimodal	✗	✓ ¹
Azure OpenAI embedding	✗	✓ ¹
Azure Machine Learning (Microsoft Foundry model catalog)	✗	✓ ¹
Document layout	✗	✓ ¹
Entity recognition	✓	✓
Image analysis ²	✓	✓
Key phrase extraction	✓	✓
Language detection	✓	✓
Text translation	✓	✗

Skill	Import data wizard	Import data (new) wizard
OCR ²	✓	✓
PII detection	✓	✗
Sentiment analysis	✓	✗
Shaper ³	✓	✗
Text Split ⁴	✓	✓
Text Merge ⁴	✓	✓

¹ This skill is available for RAG and multimodal RAG workflows only. Keyword search isn't supported.

² This skill is available for Azure Storage blobs and Microsoft OneLake files, assuming the default parsing mode. Images can be an image content type (such as PNG or JPG) or an embedded image in an application file (such as PDF).

³ This skill is added when you configure a knowledge store.

⁴ This skill is added for data chunking when you choose an embedding model. For nonembedding skills, it's added when you set the source field granularity to pages or sentences.

Semantic ranker

You can [configure semantic ranking](#) to improve the relevance of search results.

[] [Expand table](#)

Capability	Import data wizard	Import data (new) wizard
Semantic ranker	✗	✓

Knowledge store

You can [generate a knowledge store](#) for secondary storage of enriched (skills-generated) content. A knowledge store is useful for information retrieval workflows that don't require a search engine.

[] [Expand table](#)

Capability	Import data wizard	Import data (new) wizard
Knowledge store	✓	✗

What the wizards create

The following table lists the objects created by the wizards. After the objects are created, you can review their JSON definitions in the Azure portal or call them from code.

[] Expand table

Object	Description
Indexer	Configuration object that specifies a data source, target index, optional skillset, optional schedule, and optional configuration settings for error handling and base-64 encoding.
Data source	Persists connection information to a supported data source on Azure. A data source object is used exclusively with indexers.
Index	Physical data structure for full-text search, vector search, and other queries.
Skillset	(Optional) Complete set of instructions for manipulating, transforming, and shaping content, including analyzing and extracting information from image files. Skillsets are also used for integrated vectorization. If the volume of work exceeds 20 transactions per indexer per day, the skillset must include a reference to a Foundry resource that provides enrichment. For integrated vectorization, you can use either Azure Vision or an embedding model in the Foundry model catalog.
Knowledge store	(Optional) Stores enriched skillset output from tables and blobs in Azure Storage for independent analysis or downstream processing in nonsearch scenarios. Available only in the Import data wizard.

To view these objects after the wizards run:

1. Sign in to the [Azure portal](#) and select your search service.
2. From the left pane, select **Search management** to find pages for indexes, indexers, data sources, and skillsets.

Benefits

Before you write any code, you can use the wizards for prototyping and proof-of-concept testing. The wizards connect to external data sources, sample the data to create an initial index, and then import and optionally vectorize the data as JSON documents into an index on Azure AI Search.

If you're evaluating skillsets, the wizards handle output field mappings and add helper functions to create usable objects. [Text Split](#) is added when you specify a parsing mode. [Text Merge](#) is added when you choose image analysis so that the wizards can reunite text descriptions with image content. [Shaper](#) is added to support valid projections when you choose the knowledge store option. All of these tasks come with a learning curve. If you're new to enrichment, having these steps handled for you allows you to measure the value of a skill without investing much time and effort.

Sampling is the process by which an index schema is inferred, which has some limitations. When the data source is created, the wizards pick a random sample of documents to decide what columns are part of the data source. Not all files are read, as doing so could potentially take hours for large data sources. Given a selection of documents, source metadata (such as field name or type) is used to create a fields collection in an index schema. Based on the complexity of the source data, you might need to edit the initial schema for accuracy or extend it for completeness. You can make your changes inline on the index definition page.

Overall, the advantages of the wizards are clear: as long as requirements are met, you can create a queryable index within minutes. The wizards handle some of the complexities of indexing, such as serializing data as JSON documents.

Limitations

The wizards have the following limitations:

- The wizards don't support iteration or reuse. Each pass through the wizards creates an index, skillset, and indexer configuration. You can reuse data sources only in the **Import data** wizard. After you finish the wizards, you can edit the created objects by using other portal tools, the REST APIs, or the Azure SDKs.
- Source content must reside in a [supported data source](#).
- Sampling occurs over a subset of source data. For large data sources, it's possible for the wizards to miss fields. If sampling is insufficient, you might need to extend the schema or correct the inferred data types.
- [AI enrichment](#), as exposed in the Azure portal, is limited to a subset of built-in skills.
- A [knowledge store](#), which is only available through the **Import data** wizard, is limited to a few default projections and uses a default naming convention. To customize projections and names, you must create the knowledge store through the REST APIs or Azure SDKs.

Secure connections

The wizards use the Azure portal controller and public endpoints to make outbound connections. You can't use the wizards if Azure resources are accessed over a private connection or through a shared private link.

You can use the wizards over restricted public connections, but not all functionality is available.

- On supported Azure data sources protected by firewalls, you can retrieve data if you have the right firewall rules in place.

The Azure resource must admit network requests from the IP address of the device used on the connection. You should also list Azure AI Search as a trusted service on the resource's network configuration. For example, in Azure Storage, you can list

`Microsoft.Search/searchServices` as a trusted service.

- On connections to a Foundry resource for AI enrichment, or on connections to embedding models deployed in the [Foundry portal](#) or Azure OpenAI, public internet access must be enabled unless your search service meets the creation date, tier, and region requirements for private connections. For more information, see [Make outbound connections through a shared private link](#).

For AI enrichment, connections to Foundry resources are for [billing purposes](#). You're billed when API calls for built-in skills (in the **Import data** wizard or the keyword search workflow in the **Import data (new)** wizard) and integrated vectorization (in the **Import data (new)** wizard) exceed the free transaction count (20 per indexer run).

If Azure AI Search can't connect:

- In the **Import data (new)** wizard, the error is `"Access denied due to Virtual Network/Firewall rules."`.
- In the **Import data** wizard, there's no error, but the skillset won't be created.

If firewall settings prevent your wizard workflows from succeeding, consider scripted or programmatic approaches instead.

Workflow

Both wizards follow a similar high-level workflow:

1. Connect to a supported Azure data source.
2. (Optional) Add skills to extract or generate content and structure.
3. Create an index schema, inferred by sampling source data.

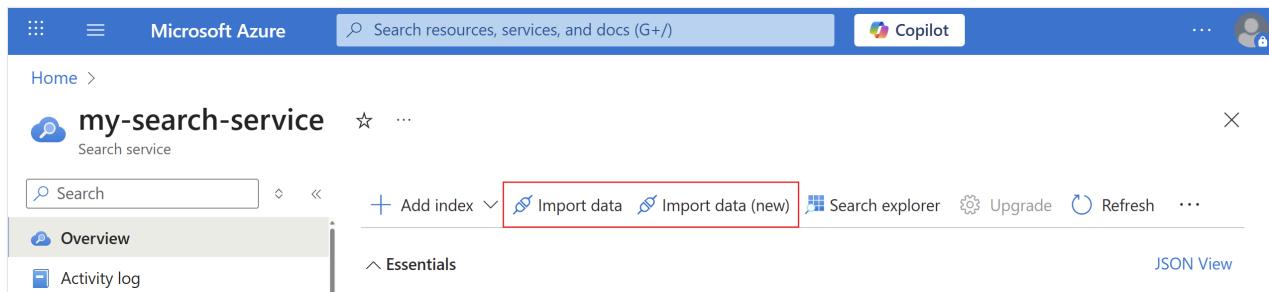
4. Run the wizard to create objects, optionally vectorize data, load data into an index, set a schedule, and configure other options.

The workflow is a one-way pipeline. You can't use the wizard to edit any of the objects that were created, but you can use other portal tools, such as the index designer, indexer designer, or JSON editors, to make allowed updates.

Starting the wizards

To start the wizards:

1. Sign in to the [Azure portal](#) and select your search service.
2. On the **Overview** page, select **Import data** or **Import data (new)**.



The wizards open fully expanded in the browser window, giving you more room to work.

3. Follow the remaining steps to create the index, indexer, and other applicable objects.

You can also launch **Import data** from other Azure services, including Azure Cosmos DB, Azure SQL Database, SQL Managed Instance, and Azure Blob Storage. Look for **Add Azure AI Search** in the left pane on the service overview page.

Data source configuration in the wizard

The wizards connect to an external [supported data source](#) using the internal logic provided by indexers, which are equipped to sample the source, read metadata, crack documents to read content and structure, and serialize contents as JSON for subsequent import to Azure AI Search.

In the **Import data** wizard, you can paste a connection to a supported data source in a different subscription or region, but the **Choose an existing connection** picker is scoped to the active subscription.

Microsoft Azure | Search resources, services, and docs (G+) ...

Dashboard > srch-eastus > Import data X

*Connect to your data Add cognitive skills (Optional) Customize target index Create an indexer

Create and load a search index using data from an external data source. Azure Cognitive Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source Azure Blob Storage

Data source name *

Data to extract Content and metadata

Parsing mode Default

Connection string * DefaultEndpointsProtocol=https;AccountName=[account...]
Choose an existing connection Under same subscription

Managed identity authentication None System-assigned User-assigned

Container name * Container name

Blob folder your/folder/here



Not all preview data sources are guaranteed to be available in the wizards. Because each data source has the potential to introduce changes downstream, a preview data source is only added when it fully supports all of the wizard's experiences, such as skillset definition and index schema inference.

You can only import from a single table, database view, or equivalent data structure. However, the structure can include hierarchical or nested substructures. For more information, see [How to model complex types](#).

Skillset configuration in the wizard

Skillset configuration occurs after the data source definition because the type of data source informs the availability of certain built-in skills. For example, if you're indexing files from Azure Blob Storage, the parsing mode you choose for those files determines whether sentiment analysis is available.

The wizards add not only skills you choose but also skills that are necessary for a successful outcome. For example, if you specify a knowledge store in the **Import data** wizard, the wizard adds a Shaper skill to support projections or physical data structures.

Skillsets are optional, and there's a button at the bottom of the page to skip ahead if you don't want AI enrichment.

Index schema configuration in the wizard

The wizards sample your data source to detect the fields and field types. Depending on the data source, they might also offer fields for indexing metadata.

Because sampling is an imprecise exercise, review the index for the following considerations:

1. Is the field list accurate? If your data source contains fields that weren't picked up in sampling, you can manually add the missed fields. You can also remove fields that don't add value to the search experience or won't be used in a [filter expression](#) or [scoring profile](#).
2. Is the data type appropriate for the incoming data? Azure AI Search supports the [entity data model \(EDM\) data types](#). For Azure SQL data, there's a [mapping chart](#) that lays out equivalent values. For more information, see [Field mappings and transformations](#).
3. Do you have one field that can serve as the *key*? This field must be an Edm.String that uniquely identifies a document. For relational data, it might be mapped to a primary key. For blobs, it might be the `metadata-storage-path`. If field values include spaces or dashes, you must set the **Base-64 Encode Key** option in the **Create an indexer** step, under **Advanced options**, to suppress the validation check for these characters.
4. Set attributes to determine how that field is used in an index.

Take your time with this step because attributes determine the physical expression of fields in the index. If you want to change attributes later, even programmatically, you almost always need to drop and rebuild the index. Core attributes like **Searchable** and **Retrievable** have a [negligible effect on storage](#). Enabling filters and using suggesters increase storage requirements.

- **Searchable** enables full-text search. Every field used in free-form queries or in query expressions must have this attribute. Inverted indexes are created for each field that you mark as **Searchable**.
- **Retrievable** returns the field in search results. Every field that provides content to search results must have this attribute. Setting this field doesn't appreciably affect index size.
- **Filterable** allows the field to be referenced in filter expressions. Every field used in a `$filter` expression must have this attribute. Filter expressions are for exact matches.

Because text strings remain intact, more storage is required to accommodate the verbatim content.

- **Facetable** enables the field for faceted navigation. Only fields also marked as **Filterable** can be marked as **Facetable**.
- **Sortable** allows the field to be used in a sort. Every field used in an `$Orderby` expression must have this attribute.

5. Do you need [lexical analysis](#)? For `Edm.String` fields that are **Searchable**, you can set an **Analyzer** if you want language-enhanced indexing and querying.

The default is *Standard Lucene*, but you can choose *Microsoft English* if you wanted to use Microsoft's analyzer for advanced lexical processing, such as resolving irregular noun and verb forms. Only language analyzers can be specified in the Azure portal. If you want to use a custom analyzer or non-language analyzer, such as Keyword or Pattern, you must create it programmatically. For more information, see [Add language analyzers](#).

6. Do you need typeahead functionality in the form of autocomplete or suggested results? Select the **Suggester** checkbox to enable [typeahead query suggestions and autocomplete](#) on selected fields. Suggesters add to the number of tokenized terms in your index and thus consume more storage.

Indexer configuration in the wizard

The last page of the wizard collects user inputs for indexer configuration. You can [specify a schedule](#) and set other options that vary by the data source type.

Internally, the wizard sets up the following definitions, which aren't visible in the indexer until after it's created.

- [Field mappings](#) between the data source and index.
- [Output field mappings](#) between the skill output and an index.

Try the wizards

The best way to understand the benefits and limitations of the wizards is to step through them. The following quickstarts are based on the wizards.

- [Quickstart: Create a search index](#)
- [Quickstart: Create a text translation and entity skillset](#)
- [Quickstart: Create a vector index](#)
- [Quickstart: Create a multimodal index](#)

Last updated on 12/05/2025

Use an Azure Logic Apps workflow for automated indexing in Azure AI Search

10/09/2025

In Azure AI Search, you can use the [Import data \(new\) wizard](#) in the Azure portal to create a logic app workflow that indexes and vectorizes your content. This capability is equivalent to an [indexer](#) and data source that generates an indexing pipeline and creates searchable content.

After you create a workflow in the wizard, you can manage the workflow in Azure Logic Apps alongside your other workflows. Behind the scenes, the wizard follows a workflow template that pulls in (ingests) content from a source for indexing in AI Search. The connectors used in this scenario are prebuilt and already exist in Azure Logic Apps, so the workflow template just provides details for those connectors to create connections to the data source, AI Search, and other items to complete the ingestion workflow.

Key features

Azure Logic Apps integration in Azure AI Search adds support for:

- [More data sources](#) from Microsoft and other providers
- Integrated vectorization
- Scheduled or on-demand indexing
- Change detection of new and existing documents

The [Import data \(new\) wizard](#) inputs include:

- A supported data source
- A supported text embedding model

After the wizard completes, you have the following components:

 Expand table

Component	Location	Description
Search index	Azure AI Search	Contains indexed content from a supported Logic Apps connector. The index schema is a default index created by the wizard. You can add extra elements, such as scoring profile or semantic configuration, but you can't change existing fields. You view, manage, and access the search index on Azure AI Search.
Logic app resource and	Azure Logic	You can view the running workflow, or you can open the designer in Azure Logic Apps to edit the workflow, as you regularly do if you'd

Component	Location	Description
workflow	Apps	started from Azure Logic Apps instead. You can edit and extend the workflow, but exercise caution so as to not break the indexing pipeline. The workflow created by the wizard uses the Consumption hosting option.
Logic app templates	Azure Logic Apps	Up to two templates created per workflow: one for on-demand indexing, and a second template for scheduled indexing. You can modify the indexing schedule in the Index multiple documents step of the workflow.

Prerequisites

Review the following requirements before you start:

- You must be an **Owner** or **Contributor** in your Azure subscription, with permissions to create resources.
- Azure AI Search, Basic pricing tier or higher if you want to use a search service identity for connections to an Azure data source, otherwise you can use any tier, subject to tier limits.
- Azure OpenAI, with a [supported embedding model](#) deployment. Vectorization is integrated into the workflow. If you don't need vectors, you can ignore the fields or try another indexing strategy.
- Azure Logic Apps is a [supported region](#). It should have a [system-assigned managed identity](#) if you want to use Microsoft Entra ID authentication on connections rather than API keys.

 **Note**

A logic app workflow is a billable resource. For more information, see [Azure Logic Apps pricing](#).

Supported regions

End-to-end functionality is available in the following regions, which provide the data source connection, document cracking, document chunks, support for Azure OpenAI embedding models, and the built-in indexing support for pulling the data. The following regions for Azure Logic Apps provide the `ParseDocument` action upon which indexing integration is based.

- East US
- East US 2

- South Central US
- West US 2
- West US 3
- Brazil South
- Australia East
- East Asia
- Southeast Asia
- North Europe
- Sweden Central
- UK South

Supported models

The logic app path through the **Import data (new)** wizard supports a selection of embedding models.

Deploy one of the following [embedding models](#) on Azure OpenAI for your end-to-end workflow.

- text-embedding-3-small
- text-embedding-3-large
- text-embedding-ada-002

Supported connectors

The following connectors are useful for indexing unstructured data, as a complement to classic indexers that primarily target structured data.

- [SharePoint](#)
- [OneDrive](#)
- [OneDrive for Business](#)
- [Azure File Storage](#)
- [Azure Queues](#)
- [Service Bus](#)

Supported actions

Logic apps integration includes the following indexing actions. For more information, see [Connect to Azure AI services from workflows in Azure Logic Apps](#).

- Check for new data.

- Get the data. An HTTP action that retrieves the uploaded document using the file URL from the trigger output.
- Compose document details. A Data Operations action that concatenates various items.
- Create token string. A Data Operations action that produces a token string using the output from the Compose action.
- Create content chunks. A Data Operations action that splits the token string into pieces, based on either the number of characters or tokens per content chunk.
- Convert tokenized data to JSON. A Data Operations action that converts the token string chunks into a JSON array.
- Select JSON array items. A Data Operations action that selects multiple items from the JSON array.
- Generate the embeddings. An Azure OpenAI action that creates embeddings for each JSON array item.
- Select embeddings and other information. A Data Operations action that selects embeddings and other document information.
- Index the data. An Azure AI Search action that indexes the data based on each selected embedding.

It also supports the following query actions:

- Wait for input prompt. A trigger that either polls or waits for new data to arrive, either based on a scheduled recurrence or in response to specific events respectively.
- Input system message for the model. A Data Operations action that provides input to train the model.
- Input sample questions and responses. A Data Operations action that provides sample customer questions and associated roles to train the model.
- Input system message for search query. A Data Operations action that provides search query input to train the model.
- Generate search query. An Inline Code action that uses JavaScript to create a search query for the vector store, based on the outputs from the preceding Compose actions.
- Convert query to embedding. An Azure OpenAI action that connects to the chat completion API, which guarantees reliable responses in chat conversations.
- Get an embedding. An Azure OpenAI action that gets a single vector embedding.
- Search the vector database. An Azure AI Search action that executes searches in the vector store.
- Create prompt. An Inline Code action that uses JavaScript to build prompts.
- Perform chat completion. An Azure OpenAI action that connects to the chat completion API, which guarantees reliable responses in chat conversations.
- Return a response. A Request action that returns the results to the caller when you use the Request trigger.

Limitations

- The search index is generated using a fixed schema (document ID, content, and vectorized content), with text extraction only. You can [modify the index](#) as long as the update doesn't affect existing fields.
- Vectorization supports text embedding only.
- Deletion detection isn't supported. You must manually [delete orphaned documents](#) from the index.
- Duplicate documents in the search index are a known issue in this preview. Consider deleting objects and starting over if this becomes an issue.
- No support for private endpoints in the logic app workflow created by the portal wizard. The workflow is hosted using the [Consumption hosting option](#) and is subject to its constraints. To use the [Standard](#) hosting option, use a programmatic approach to creating the workflow.
- All actions are generally available except for

Create a logic app workflow

Follow these steps to create a logic app workflow for indexing content in Azure AI Search.

1. Start the [Import data \(new\)](#) wizard in the Azure portal.
2. Choose a [supported Azure Logic Apps connector](#).

Home > Quickstart wizard ...

Let's start by picking a data source... Available options are listed below. [Learn more](#)

Filter by name...

Azure Blob Storage Built-in indexer	Azure Data Lake Storage Gen2 Built-in indexer	Azure Cosmos DB Built-in indexer
Azure SQL Database Built-in indexer	Azure Table Storage Built-in indexer	Fabric OneLake files (Preview) Built-in indexer
SharePoint Logic Apps indexer	OneDrive Logic Apps indexer	OneDrive for Business Logic Apps indexer
Azure File Storage Logic Apps indexer	Azure Queues Logic Apps indexer	Service Bus Logic Apps indexer
Amazon S3 Logic Apps indexer	Dropbox Logic Apps indexer	SFTP - SSH Logic Apps indexer

3. In **Connect to your data**, provide a name prefix used for the search index and workflow.
Having a common name helps you manage them together.
4. Specify the indexing frequency. If you choose on a schedule, a template that includes a scheduling option is used to create the workflow. You can modify the indexing schedule in the **Index multiple documents** step of the workflow after it's created.
5. Select an authentication type where the logic app workflow connects to the search engine and starts the indexing process. The workflow can connect using [Azure AI Search API keys](#) or the wizard can create a role assignment that grants permissions to the Logic Apps system-assigned managed identity, assuming one exists.
6. Select **Next** to continue to the next page.
7. In **Vectorize your text**, provide the model deployment and Azure OpenAI connection information. Choose the subscription and service, a [supported text embedding model](#), and the authentication type that the workflow uses to connect to Azure OpenAI.
8. Select **Next** to continue to the next page. Review the configuration.
9. Select **Create** to begin processing.

The workflow runs as a serverless workflow in Logic Apps (Consumption), separate from the AI Search service.

10. Confirm index creation in the Azure portal, in the **Indexes** page in Azure AI Search. [Search Explorer](#) is the first tab. Select **Search** to return some content.

Modify existing objects

You can make the following modifications to a search index without breaking indexing:

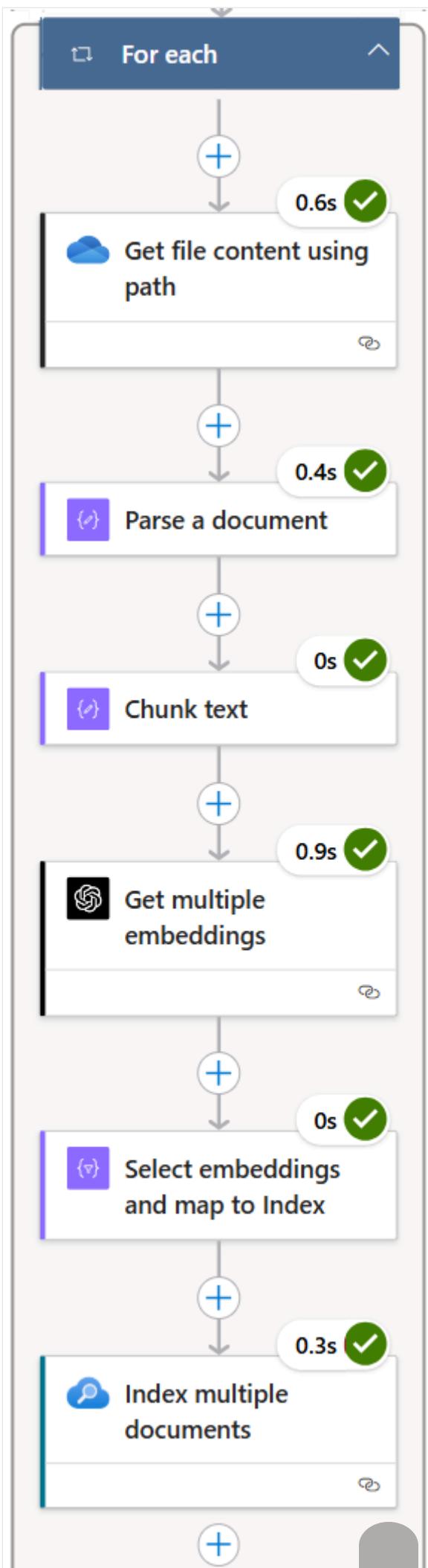
- [Add scoring profiles](#)
- [Add semantic ranking](#)
- [Add spell check](#)
- [Add synonym maps](#)
- [Add suggesters](#)

You can make the following updates to a workflow without breaking indexing:

- Modify **List files in folder** to change the number of documents sent to indexing.
- Modify **Chunk Text** to vary token inputs. The recommended token size is 512 tokens for most scenarios.
- Modify **Chunk Text** to add a page overlap length.

- Modify **Index multiple documents** step to control indexing frequency if you chose scheduled indexing in the wizard.

In logic apps designer, review the workflow and each step in the indexing pipeline. The workflow specifies document extraction, default document chunking ([Text Split skill](#)), embedding ([Azure OpenAI embedding skill](#)), output field mappings, and finally indexing.





Template and workflow management

The wizard creates templates and workflows when you specify a Logic Apps indexer. To create and manage them, including template deletion, use the logic app designer. The Azure portal search service dashboard doesn't provide template or workflow management, and currently there's no programmatic support in Azure AI Search APIs.

Related content

- [Indexers](#)
- [Connect to Azure AI services from workflows in Azure Logic Apps](#)
- [Manage logic apps](#)

Create an indexer in Azure AI Search

06/17/2025

This article focuses on the basic steps of creating an indexer that's used to automate data ingestion for supported data sources. Depending on the data source and your workflow, more configuration might be necessary.

You can use an indexer to automate data import and indexing in Azure AI Search. An indexer is a named object on a search service that connects to an external Azure data source, reads and serializes the data, and passes it to a search engine for indexing. Using indexers significantly reduces the quantity and complexity of the code you need to write if you're using a supported data source.

Indexers support two workflows:

- **Raw content indexing (plain text or vectors):** Extract strings and metadata from textual content used for full text search scenarios. Extracts raw vector content used for vector search (for example, vectors in an Azure SQL database or Azure Cosmos DB collection). In this workflow, indexing occurs only over existing content that you provide.
- **Skills-based indexing:** Extends indexing through built-in or custom skills that create or generate new searchable content. For example, you can add integrated machine learning for analysis over images and unstructured text, extracting or inferring text and structure. Or, use skills to chunk and vectorize content from text and images. Skills-based indexing creates or generates new content that doesn't exist in your external data source. New content becomes part of your index when you add fields to the index schema that accepts the incoming data. To learn more, see [AI enrichment in Azure AI Search](#).

Prerequisites

- A [supported data source](#) that contains the content you want to ingest.
- An [indexer data source](#) that sets up a connection to external data.
- A [search index](#) that can accept incoming data.
- Be under the [maximum limits](#) for your service tier. The Free tier allows three objects of each type and 1-3 minutes of indexer processing, or 3-10 minutes if there's a skillset.

Indexer patterns

When you create an indexer, the definition is one of two patterns: *content-based indexing* or *skills-based indexing*. The patterns are the same, except that skills-based indexing has more definitions.

Indexer example for content-based indexing

Content-based indexing for full text or vector search is the primary use case for indexers. For this workflow, an indexer looks like this example.

JSON

```
{  
  "name": (required) String that uniquely identifies the indexer,  
  "description": (optional),  
  "dataSourceName": (required) String indicating which existing data source to  
use,  
  "targetIndexName": (required) String indicating which existing index to use,  
  "parameters": {  
    "batchSize": null,  
    "maxFailedItems": 0,  
    "maxFailedItemsPerBatch": 0,  
    "base64EncodeKeys": false,  
    "configuration": {}  
  },  
  "fieldMappings": (optional) unless field discrepancies need resolution,  
  "disabled": null,  
  "schedule": null,  
  "encryptionKey": null  
}
```

Indexers have the following requirements:

- A `name` property that uniquely identifies the indexer in the indexer collection
- A `dataSourceName` property that points to a data source object. It specifies a connection to external data
- A `targetIndexName` property that points to the destination search index

Other parameters are optional and modify run time behaviors, such as how many errors to accept before failing the entire job. Required parameters are specified in all indexers and are documented in the [REST API reference](#).

Data source-specific indexers for blobs, SQL, and Azure Cosmos DB provide extra `configuration` parameters for source-specific behaviors. For example, if the source is Blob Storage, you can set a parameter that filters on file extensions, such as:

JSON

```
"parameters" : { "configuration" : { "indexedFileNameExtensions" : ".pdf,.docx" } }
```

If the source is Azure SQL, you can set a query time-out parameter.

[Field mappings](#) are used to explicitly map source-to-destination fields if there are discrepancies by name or type between a field in the data source and a field in the search index.

By default, an indexer runs immediately when you create it on the search service. If you don't want indexer execution, set `disabled` to `true` when creating the indexer.

You can also [specify a schedule](#) or set an [encryption key](#) for supplemental encryption of the indexer definition.

Indexer example for skills-based indexing

Skills-based indexing uses [AI enrichment](#) to process content that isn't searchable in its raw form. All of the above properties and parameters apply, but the following extra properties are specific to AI enrichment: `skillSetName`, `cache`, `outputFieldMappings`.

JSON

```
{
  "name": (required) String that uniquely identifies the indexer,
  "dataSourceName": (required) String, provides raw content that will be enriched,
  "targetIndexName": (required) String, name of an existing index,
  "skillsetName" : (required for AI enrichment) String, name of an existing
skillset,
  "cache": {
    "storageConnectionString" : (required if you enable the cache) Connection
string to a blob container,
    "enableReprocessing": true
  },
  "parameters": { },
  "fieldMappings": (optional) Maps fields in the underlying data source to fields
in an index,
  "outputFieldMappings" : (required) Maps skill outputs to fields in an index,
}
```

AI enrichment is its own subject area and is out of scope for this article. For more information, start with [AI enrichment](#), [Skillsets in Azure AI Search](#), [Create a skillset](#), [Map enriched output fields](#), and [Enable caching for AI enrichment](#).

Prepare external data

Indexers work with data sets. When you run an indexer, it connects to your data source, retrieves the data from the container or folder, optionally serializes it into JSON before passing it to the search engine for indexing. This section describes the requirements of incoming data for text-based indexing.

[] Expand table

Source	Tasks
data	
JSON documents	JSON documents can contain text, numbers, and vectors. Make sure the structure or shape of incoming data corresponds to the schema of your search index. Most search indexes are fairly flat, where the fields collection consists of fields at the same level. However, hierarchical or nested structures are possible through complex fields and collections .
Relational	<p>Provide data as a flattened row set, where each row becomes a full or partial search document in the index.</p> <p>To flatten relational data into a row set, you should create a SQL view, or build a query that returns parent and child records in the same row. For example, the built-in hotels sample dataset is an SQL database that has 50 records (one for each hotel), linked to room records in a related table. The query that flattens the collective data into a row set embeds all of the room information in JSON documents in each hotel record. The embedded room information is generated by a query that uses a FOR JSON AUTO clause.</p> <p>You can learn more about this technique in define a query that returns embedded JSON. This is just one example; you can find other approaches that produce the same result.</p>
Files	An indexer generally creates one search document for each file, where the search document consists of fields for content and metadata. Depending on the file type, the indexer can sometimes parse one file into multiple search documents . For example, in a CSV file, each row can become a standalone search document.

Remember that you only need to pull in searchable and filterable data:

- Searchable data is text or vectors
- Filterable data is text and numbers (non-vector fields)

Azure AI Search can't do a full-text search over binary data in any format, although it can extract and infer text descriptions of image files (see [AI enrichment](#)) to create searchable content. Likewise, large text can be broken down and analyzed by natural language models to find structure or relevant information, generating new content that you can add to a search document. It can also do vector search over embeddings, including quantized embeddings in a binary format.

Given that indexers don't fix data problems, other forms of data cleansing or manipulation might be needed. For more information, you should refer to the product documentation of

your Azure database product.

Prepare a data source

Indexers require a data source that specifies the type, container, and connection.

1. Make sure you're using a [supported data source type](#).
2. [Create a data source](#) definition. The following data sources are a few of the more frequently used sources:
 - [Azure Blob Storage](#)
 - [Azure Cosmos DB](#)
 - [Azure SQL Database](#)
3. If the data source is a database, such as Azure SQL or Cosmos DB, enable change tracking. Azure Storage has built-in change tracking through the `LastModified` property on every blob, file, and table. The links for the various data sources explain which change tracking methods are supported by indexers.

Prepare an index

Indexers also require a search index. Recall that indexers pass data off to the search engine for indexing. Just as indexers have properties that determine execution behavior, an index schema has properties that profoundly affect how strings are indexed (only strings are analyzed and tokenized).

1. Start with [Create a search index](#).
2. Set up the fields collection and field attributes.

Fields are the only receptors of external content. Depending on how the fields are attributed in the schema, the values for each field are analyzed, tokenized, or stored as verbatim strings for filters, fuzzy search, and typeahead queries.

Indexers can automatically map source fields to target index fields when the names and types are equivalent. If a field can't be implicitly mapped, remember that you can [define an explicit field mapping](#) that tells the indexer how to route the content.

3. Review the analyzer assignments on each field. Analyzers can transform strings. As such, indexed strings might be different from what you passed in. You can evaluate the effects of analyzers using [Analyze Text \(REST\)](#). For more information about analyzers, see [Analyzers for text processing](#).

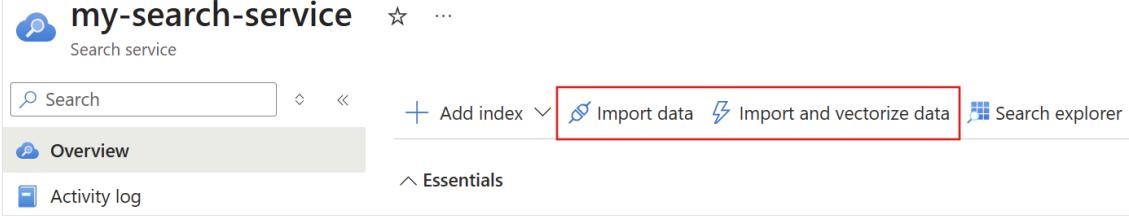
During indexing, an indexer only checks field names and types. There's no validation step that ensures incoming content is correct for the corresponding search field in the index.

Create an indexer

When you're ready to create an indexer on a remote search service, you need a search client. A search client can be the Azure portal, a REST client, or code that instantiates an indexer client. We recommend the Azure portal or REST APIs for early development and proof-of-concept testing.

Azure portal

1. Sign in to the [Azure portal](#) and select your search service.
2. Choose from the following options:
 - **Import wizards:** The wizards are unique in that they create all of the required elements. Other approaches require a predefined data source and index.
 - **Add indexer:** A visual editor for specifying an indexer definition.



The screenshot shows the Azure portal interface for a search service named 'my-search-service'. The 'Indexers' blade is open. On the left, there's a sidebar with various navigation links. The 'Indexers' link is currently selected and highlighted in grey. At the top right, there are buttons for 'Add indexer' (which is highlighted with a red box), 'Refresh', and 'Delete'. Below the top bar, there's a search bar and a 'Filter by name...' input field. The main content area displays a table with columns for 'Status' and 'Name', though no specific data rows are visible.

Run the indexer

By default, an indexer runs immediately when you create it on the search service. You can override this behavior by setting `disabled` to `true` in the indexer definition. Indexer execution is the moment of truth where you find out if there are problems with connections, field mappings, or skillset construction.

There are several ways to run an indexer:

- Run on indexer creation or update (default).
- Run on demand when there are no changes to the definition, or precede with `reset` for full indexing. For more information, see [Run or reset indexers](#).
- [Schedule indexer processing](#) to invoke execution at regular intervals.

Scheduled execution is usually implemented when you have a need for incremental indexing so that you can pick up the latest changes. As such, scheduling has a dependency on change detection.

Indexers are one of the few subsystems that make overt outbound calls to other Azure resources. In terms of Azure roles, indexers don't have separate identities; a connection from the search engine to another Azure resource is made using the [system or user-assigned managed identity](#) of a search service. If the indexer connects to an Azure resource on a virtual

network, you should create a [shared private link](#) for that connection. For more information about secure connections, see [Security in Azure AI Search](#).

Check results

[Monitor indexer status](#) to check for status. Successful execution can still include warning and notifications. Be sure to check both successful and failed status notifications for details about the job.

For content verification, [run queries](#) on the populated index that return entire documents or selected fields.

Change detection and internal state

If your data source supports change detection, an indexer can detect underlying changes in the data and process just the new or updated documents on each indexer run, leaving unchanged content as-is. If indexer execution history says that a run was successful with *0/0* documents processed, it means that the indexer didn't find any new or changed rows or blobs in the underlying data source.

Change detection logic is built into the data platforms. How an indexer supports change detection varies by data source:

- Azure Storage has built-in change detection, which means an indexer can recognize new and updated documents automatically. Blob Storage, Azure Table Storage, and Azure Data Lake Storage Gen2 stamp each blob or row update with a date and time. An indexer automatically uses this information to determine which documents to update in the index. For more information about deletion detection, see [Change and delete detection using indexers for Azure Storage](#).
- Cloud database technologies provide optional change detection features in their platforms. For these data sources, change detection isn't automatic. You need to specify in the data source definition which policy is used:
 - [Azure SQL \(change detection\)](#)
 - [Azure DB for MySQL \(change detection\)](#)
 - [Azure Cosmos DB for NoSQL \(change detection\)](#)
 - [Azure Cosmos DB for MongoDB \(change detection\)](#)
 - [Azure Cosmos DB for Apache Gremlin \(change detection\)](#)

Indexers keep track of the last document it processed from the data source through an internal *high water mark*. The marker is never exposed in the API, but internally the indexer keeps track

of where it stopped. When indexing resumes, either through a scheduled run or an on-demand invocation, the indexer references the high water mark so that it can pick up where it left off.

If you need to clear the high water mark to reindex in full, you can use [Reset Indexer](#). For more selective reindexing, use [Reset Skills](#) or [Reset Documents](#). Through the reset APIs, you can clear internal state, and also flush the cache if you enabled [incremental enrichment](#). For more background and comparison of each reset option, see [Run or reset indexers, skills, and documents](#).

Related content

- [Index data from Azure Blob Storage](#)
- [Index data from Azure SQL database](#)
- [Index data from Azure Data Lake Storage Gen2](#)
- [Index data from Azure Table Storage](#)
- [Index data from Azure Cosmos DB](#)

Run or reset indexers, skills, or documents

10/02/2025

In Azure AI Search, there are several ways to run an indexer:

- [Run immediately upon indexer creation](#). This is the default unless you create the indexer in a "disabled" state.
- [Run on a schedule](#) to invoke execution at regular intervals.
- Run on demand, with or without a "reset".

This article explains how to run indexers on demand, with and without a reset. It also describes indexer execution, duration, and concurrency.

How indexers connect to Azure resources

Indexers are one of the few subsystems that make overt outbound calls to other Azure resources. Depending on the external data source, you can use keys or roles to authenticate the connection.

In terms of Azure roles, indexers don't have separate identities: a connection from the search engine to another Azure resource is made using the [system or user-assigned managed identity](#) of a search service, plus a role assignment on the target Azure resource. If the indexer connects to an Azure resource on a virtual network, you should create a [shared private link](#) for that connection.

Indexer execution

A search service runs one indexer job per [search unit](#). Every search service starts with one search unit, but each new partition or replica increases the search units of your service. You can check the search unit count in the Azure portal's Essential section of the [Overview](#) page. If you need concurrent processing, make sure your search units include sufficient replicas. Indexers don't run in the background, so you might experience more query throttling than usual if the service is under pressure.

The following screenshot shows the number of search units, which determines how many indexers can run at once.

Essentials		View Cost	JSON View
Resource group (move) :		Url	: https://demo-srch.search.windows.net
Location (move)	: South Central US	Pricing tier	: Basic
Subscription (move)	:	Replicas	: 2 (99.9% read SLA)
Subscription ID	:	Partitions	: 1
Status	: Running	Search units : 2	

Once indexer execution starts, you can't pause or stop it. Indexer execution stops when there are no more documents to load or refresh, or when the [maximum running time limit](#) is reached.

You can run multiple indexers at one time assuming sufficient capacity, but each indexer itself is single-instance. Starting a new instance while the indexer is already in execution produces this error: "Failed to run indexer "<indexer name>" error: "Another indexer invocation is currently in progress; concurrent invocations are not allowed."

Indexer execution environment

An indexer job runs in a managed execution environment. Currently, there are two environments:

- A private execution environment runs on search clusters that are specific to your search service.
- A multitenant environment has content processors that are managed and secured by Microsoft at no extra cost. This environment is used to offload computationally intensive processing, leaving service-specific resources available for routine operations. Whenever possible, most skillsets execute in the multitenant environment. This is the default.

Computationally intensive processing refers to skillsets running on content processors and indexer jobs that process a high volume of documents, or documents of a large size. Non-skillset processing on the multitenant content processors is determined by heuristics and system information and isn't under customer control.

You can prevent usage of the multitenant environment on Standard2 or higher services by pinning an indexer and skillset processing exclusively to your search clusters. [Set the executionEnvironment parameter](#) in the indexer definition to always run an indexer in the private execution environment.

[IP firewalls](#) block the multitenant environment, so if you have a firewall, [create a rule](#) that allows multitenant processor connections.

Indexer limits vary for each environment:

Workload	Maximum duration	Maximum jobs	Execution environment
Private execution	24 hours	One indexer job per search unit ¹	Indexing doesn't run in the background. Instead, the search service balances all indexing jobs against ongoing queries and object management actions (such as creating or updating indexes). When running indexers, you should expect to see some query latency if indexing volumes are large.
Multitenant	2 hours ²	Indeterminate ³	Because the content processing cluster is multitenant, content processors are added to meet demand. If you experience a delay in on-demand or scheduled execution, it's probably because the system is either adding processors or waiting for one to become available.

¹ Search units can be [flexible combinations](#) of partitions and replicas, but indexer jobs aren't tied to one or the other. In other words, if you have 12 units, you can have 12 indexer jobs running concurrently in private execution, no matter how the search units are deployed.

² If more than two hours are needed to process all of the data, [enable change detection](#) and [schedule the indexer](#) to run at 5-minute intervals to resume indexing quickly if it stops due to a time out. See [Indexing a large data set](#) for more strategies.

³ "Indeterminate" means that the limit isn't quantified by the number of jobs. Some workloads, such as skillset processing, can run in parallel, which could result in many jobs even though only one indexer is involved. Although the environment doesn't impose constraints, [indexer limits](#) for your search service still apply.

Run without reset

A [Run Indexer](#) operation detects and processes only what it necessary to synchronize the search index with changes in the underlying data source. Incremental indexing starts by locating an internal high-water mark to find the last updated search document, which becomes the starting point for indexer execution over new and updated documents in the data source.

[Change detection](#) is essential for determining what's new or updated in the data source. Indexers use the change detection capabilities of the underlying data source to determine what's new or updated in the data source.

- Azure Storage has built-in change detection through its LastModified property.

- Other data sources, such as Azure SQL or Azure Cosmos DB, have to be configured for change detection before the indexer can read new and updated rows.

If the underlying content is unchanged, a run operation has no effect. In this case, indexer execution history indicates `0\0` documents processed.

You need to reset the indexer, as explained in the next section, to reprocess in full.

Resetting indexers

After the initial run, an indexer keeps track of which search documents are indexed through an internal *high-water mark*. The marker is never exposed, but internally the indexer knows where it last stopped.

If you need to rebuild all or part of an index, use Reset APIs available at decreasing levels in the object hierarchy:

- [Reset Indexers](#) clears the high-water mark and performs a full reindex of all documents
- [Resync Indexers \(preview\)](#) performs an efficient partial reindex of all documents
- [Reset Documents \(preview\)](#) reindexes a specific document or list of documents
- [Reset Skills \(preview\)](#) invokes skill processing for a specific skill

After reset, follow with a Run command to reprocess new and existing documents. Orphaned search documents having no counterpart in the data source can't be removed through reset/run. If you need to delete documents, see [Documents - Index](#) instead.

(!) Note

Tables can't be empty. If you use TRUNCATE TABLE to clear rows, a reset and rerun of the indexer won't remove the corresponding search documents. To remove orphaned search documents, you must [index them with a delete action](#).

How to reset and run indexers

Reset clears the high-water mark. All documents in the search index are flagged for full overwrite, without inline updates or merging into existing content. For indexers with a skillset and [enrichment caching](#), resetting the index also implicitly resets the skillset.

The actual work occurs when you follow a reset with a Run command:

- All new documents found the underlying source are added to the search index.

- All documents that exist in both the data source and search index are overwritten in the search index.
- Any enriched content created from skillsets are rebuilt. The enrichment cache, if one is enabled, is refreshed.

As previously noted, reset is a passive operation: you must follow with a Run request to rebuild the index.

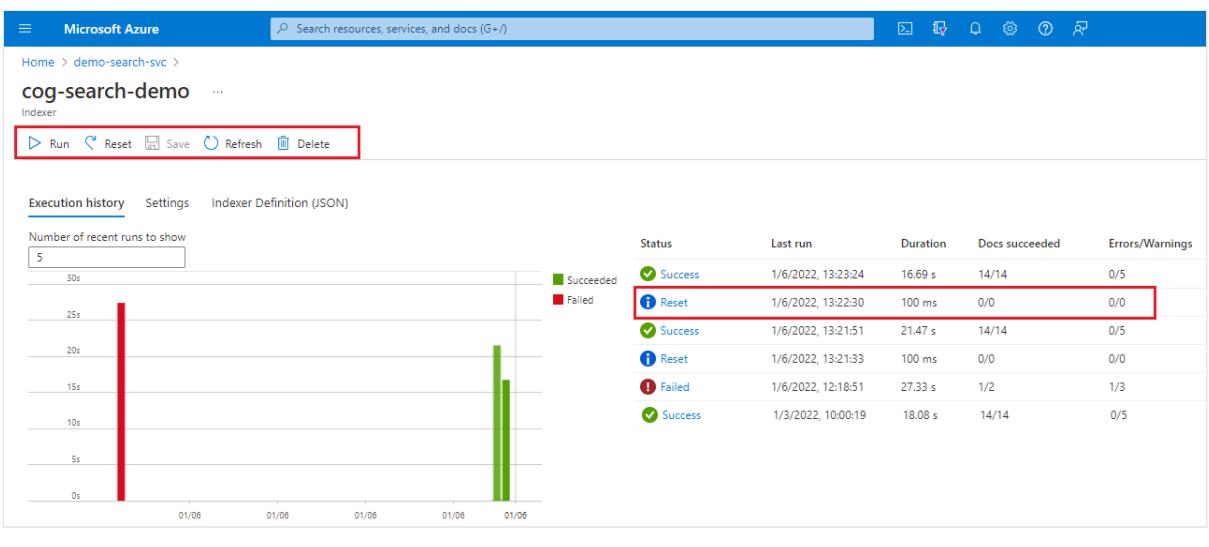
Reset/run operations apply to a search index or a knowledge store, to specific documents or projections, and to cached enrichments if a reset explicitly or implicitly includes skills.

Reset also applies to create and update operations. It won't trigger deletion or clean up of orphaned documents in the search index. For more information about deleting documents, see [Documents - Index](#).

Once you reset an indexer, you can't undo the action.

Azure portal

1. Sign in to the [Azure portal](#) and open the search service page.
2. On the **Overview** page, select the **Indexers** tab.
3. Select an indexer.
4. Select the **Reset** command, and then select **Yes** to confirm the action.
5. Refresh the page to show the status. You can select the item to view its details.
6. Select **Run** to start indexer processing, or wait for the next scheduled execution.



The screenshot shows the Azure portal interface for a search service named "demo-search-svc". Under the "Indexers" tab, there is a single indexer named "cog-search-demo". The "Run" button is highlighted with a red box. Below the indexer name, there's a "Execution history" section with a bar chart showing recent run durations (0s, 5s, 10s, 15s, 20s, 25s, 30s) and a table of execution logs. The log table includes columns for Status, Last run, Duration, Docs succeeded, and Errors/Warnings. One entry for a "Reset" operation is highlighted with a red box.

Status	Last run	Duration	Docs succeeded	Errors/Warnings
✓ Success	1/6/2022, 13:23:24	16.69 s	14/14	0/5
↻ Reset	1/6/2022, 13:22:30	100 ms	0/0	0/0
✓ Success	1/6/2022, 13:21:51	21.47 s	14/14	0/5
↻ Reset	1/6/2022, 13:21:33	100 ms	0/0	0/0
⚠ Failed	1/6/2022, 12:18:51	27.33 s	1/2	1/3
✓ Success	1/3/2022, 10:00:19	18.08 s	14/14	0/5

How to reset skills (preview)

The Reset Skills request selectively processes one or more skills on the next indexer run. For indexers that have skillsets, you can reset individual skills to force reprocessing of just that skill and any downstream skills that depend on its output. The [enrichment cache](#), if you enabled it, is also refreshed.

For indexers that have caching enabled, you can explicitly request processing for skill updates that the indexer cannot detect. For example, if you make external changes, such as revisions to a custom skill, you can use this API to rerun the skill. Outputs, such as a knowledge store or search index, are refreshed using reusable data from the cache and new content per the updated skill.

We recommend the [latest preview API](#).

HTTP

```
POST /skillsets/[skillset name]/resetskills?api-version=2025-08-01-preview
{
    "skillNames" : [
        "#1",
        "#5",
        "#6"
    ]
}
```

You can specify individual skills, as indicated in the example above, but if any of those skills require output from unlisted skills (#2 through #4), unlisted skills will run unless the cache can provide the necessary information. In order for this to be true, cached enrichments for skills #2 through #4 must not have dependency on #1 (listed for reset).

If no skills are specified, the entire skillset is executed and if caching is enabled, the cache is also refreshed.

Remember to follow up with [Run Indexer](#) to invoke actual processing.

How to reset docs (preview)

The [Indexers - Reset Docs \(preview\)](#) accepts a list of document keys so that you can refresh specific documents. If specified, the reset parameters become the sole determinant of what gets processed, regardless of other changes in the underlying data. For example, if 20 blobs were added or updated since the last indexer run, but you only reset one document, only that document is processed.

On a per-document basis, all fields in the search document are refreshed with values and metadata from the data source. You can't pick and choose which fields to refresh.

If the data source is Azure Data Lake Storage (ADLS) Gen2, and the blobs are associated with permission metadata, those permissions are also re-ingested in the search index if permissions change in the underlying data. For more information, see [Re-indexing ACL and RBAC scope with ADLS Gen2 indexers](#).

If the document is enriched through a skillset and has cached data, the skillset is invoked for just the specified documents, and the cache is updated for the reprocessed documents.

When you're testing this API for the first time, the following APIs can help you validate and test the behaviors. We recommend the latest preview API.

1. Call [Indexers - Get Status](#) with a preview API version to check reset status and execution status. You can find information about the reset request at the end of the status response.
2. Call [Indexers - Reset Docs](#) with a preview API version to specify which documents to process.

HTTP

```
POST https://[service name].search.windows.net/indexers/[indexer
name]/resetdocs?api-version=2025-08-01-preview
{
    "documentKeys" : [
        "1001",
        "4452"
    ]
}
```

- The API accepts two types of document identifiers as input: Document keys that uniquely identify documents in a search index, and datasource document identifiers that uniquely identify documents in a data source. The body should contain either a list of document keys or a list of data source document identifiers that the indexer looks for in the data source. Invoking the API adds the document keys or data source document identifiers to be reset to the indexer metadata. On the next scheduled or on-demand run of the indexer, the indexer processes only the reset documents.
- If you use document keys to reset documents and your document keys are referenced in an indexer field mapping, the indexer uses field mapping to locate the appropriate field in the underlying data source.

- The document keys provided in the request are values from the search index, which can be different from the corresponding fields in the data source. If you're unsure of the key value, [send a query](#) to return the value. You can use `select` to return just the document key field.
- For blobs that are parsed into multiple search documents (where parsingMode is set to `jsonLines` or `jsonArrays`, or `delimitedText`), the document key is generated by the indexer and might be unknown to you. In this scenario, a query for the document key to return the correct value.
- If you want the indexer to stop trying to process reset documents, you can set "documentKeys" or "datasourceDocumentIds" to an empty list "[]". This results in the indexer resuming regular indexing based on the high water mark. Invalid document keys or document keys that don't exist are ignored.

3. Call [Run Indexer](#) (any API version) to process the documents you specified. Only those specific documents are indexed.

4. Call [Run Indexer](#) a second time to process from the last high-water mark.

5. Call [Search Documents](#) to check for updated values, and also to return document keys if you're unsure of the value. Use `"select": "<field names>"` if you want to limit which fields appear in the response.

Overwriting the document key list

Calling Reset Documents API multiple times with different keys appends the new keys to the list of document keys reset. Calling the API with the `overwrite` parameter set to true will overwrite the current list with the new one:

HTTP

```
POST https://[service name].search.windows.net/indexers/[indexer name]/resetdocs?
api-version=2025-08-01-Preview
{
  "documentKeys" : [
    "200",
    "630"
  ],
  "overwrite": true
}
```

How to resync indexers (preview)

[Resync Indexers](#) is a preview REST API that performs a partial reindex of all documents. An indexer is considered synchronized with its data source when specific fields of all documents in the target index are consistent with the data in the data source. Typically, an indexer achieves synchronization after a successful initial run. If a document is deleted from the data source, the indexer remains synchronized according to this definition. However, during the next indexer run, the corresponding document in the target index will be removed if delete tracking is enabled.

If a document is modified in the data source, the indexer becomes unsynchronized. Generally, change tracking mechanisms will resynchronize the indexer during the next run. For example, in Azure Storage, modifying a blob updates its last modified time, allowing it to be re-indexed in the subsequent indexer run because the updated time surpasses the high-water mark set by the previous run.

In contrast, for certain data sources like ADLS Gen2, altering the Access Control Lists (ACLs) of a blob does not change its last modified time, rendering change tracking ineffective if ACLs are to be ingested. Consequently, the modified blob will not be re-indexed in the subsequent run, as only documents modified after the last high-water mark are processed.

While using either "reset" or "reset docs" can address this issue, "reset" can be time-consuming and inefficient for large datasets, and "reset docs" requires identifying the document key of the blob intended for update.

Resync Indexers offers an efficient and convenient alternative. Users simply place the indexer in resync mode and specify the content to resynchronize by calling the resync indexers API. In the next run, the indexer will inspect only relevant portion of data in the source and avoid any unnecessary processing that is unrelated to the specified data. It will also query the existing documents in the target index and only update the documents that show discrepancies between the data source and the target index. After the resync run, the indexer will be synchronized and revert to regular indexer run mode for subsequent runs.

How to resync and run indexers

1. Call [Indexers - Resync](#) with a preview API version to specify what content to re-synchronize.

HTTP

```
POST https://[service name].search.windows.net/indexers/[indexer  
name]/resync?api-version=2025-08-01-preview  
{  
    "options" : [  
        "permissions"  
    ]  
}
```

```
    ]  
}
```

- The `options` field is required. Currently the only supported option is `permissions`. That is, only permission filter fields in the target index will be updated.

2. Call [Run Indexer](#) (any API version) to re-synchronize the indexer.
3. Call [Run Indexer](#) a second time to process from the last high-water mark.

Check reset status "currentState"

To check reset status and to see which document keys are queued up for processing, following these steps.

1. Call [Get Indexer Status](#) with a preview API.

The preview API will return the `currentState` section, found at the end of the response.

JSON

```
"currentState": {  
    "mode": "indexingResetDocs",  
    "allDocsInitialTrackingState": "{\"LastFullEnumerationStartTime\":\"2021-02-06T19:02:07.0323764+00:00\", \"LastAttemptedEnumerationStartTime\":\"2021-02-06T19:02:07.0323764+00:00\", \"NameHighWaterMark\":null}",  
    "allDocsFinalTrackingState": "{\"LastFullEnumerationStartTime\":\"2021-02-06T19:02:07.0323764+00:00\", \"LastAttemptedEnumerationStartTime\":\"2021-02-06T19:02:07.0323764+00:00\", \"NameHighWaterMark\":null}",  
    "resetDocsInitialTrackingState": null,  
    "resetDocsFinalTrackingState": null,  
    "resyncInitialTrackingState": null,  
    "resyncFinalTrackingState": null,  
    "resetDocumentKeys": [  
        "200",  
        "630"  
    ]  
}
```

2. Check the "mode":

For Reset Skills, "mode" should be set to `indexingAllDocs` (because potentially all documents are affected, in terms of the fields that are populated through AI enrichment).

For Resync Indexers, "mode" should be set to `indexingResync`. The indexer checks all documents and focuses on interested data in data source and interested fields in the target index.

For Reset Documents, "mode" should be set to `indexingResetDocs`. The indexer retains this status until all the document keys provided in the reset documents call are processed, during which time no other indexer jobs will execute while the operation is progressing. Finding all of the documents in the document keys list requires cracking each document to locate and match on the key, and this can take a while if the data set is large. If a blob container contains hundreds of blobs, and the docs you want to reset are at the end, the indexer won't find the matching blobs until all of the others have been checked first.

3. After the documents are reprocessed, run Get Indexer Status again. The indexer returns to the `indexingAllDocs` mode and will process any new or updated documents on the next run.

Check indexer runtime quota for S3 HD search services

Applies to search services at the Standard 3 High Density (S3 HD) pricing tier.

To help you monitor indexer running times relative to the 24-hour window, [Get Service Statistics](#) and [Get Indexer Status](#) now return more information in the response.

Track cumulative runtime quota

Track a search service's cumulative indexer runtime usage and determine how much runtime quota is left within the current 24-hour window period.

Send a GET request to the search service resource provider. For help with setting up a REST client and getting an access token, see [Connect to a search service](#).

HTTP

```
GET {{search-endpoint}}/servicestats?api-version=2025-08-01-preview
Content-Type: application/json
Authorization: Bearer {{accessToken}}
```

Responses include `indexersRuntime` properties, showing start and end times, seconds used, seconds remaining, and cumulative runtime within the last 24 hours.

Track indexer runtime quota

Return the same information for a single indexer.

HTTP

```
GET {{search-endpoint}}/indexers/hotels-sample-indexer/search.status?api-version=2025-08-01-preview
Content-Type: application/json
Authorization: Bearer {{accessToken}}
```

Responses include a `runtime` properties, showing start and end times, seconds used, and seconds remaining.

Next steps

Reset APIs are used to inform the scope of the next indexer run. For actual processing, you'll need to invoke an on-demand indexer run or allow a scheduled job to complete the work. After the run is finished, the indexer returns to normal processing, whether that is on a schedule or on-demand processing.

After you reset and rerun indexer jobs, you can monitor status from the search service, or obtain detailed information through resource logging.

- [Monitor search indexer status](#)
- [Collect and analyze log data](#)
- [Schedule an indexer](#)

Schedule an indexer in Azure AI Search

09/24/2025

Indexers can be configured to run on a schedule when you set the `schedule` property. Some situations where indexer scheduling is useful include:

- Source data is changing over time, and you want the indexer to automatically process the difference.
- Source data is very large, and you need a recurring schedule to index all of the content.
- An index is populated from multiple sources, using multiple indexers, and you want to stagger the jobs to reduce conflicts.

When indexing can't complete within the [typical 2-hour processing window](#), you can schedule the indexer to run on a 2-hour cadence to work through a large volume of data. As long as your data source supports [change detection logic](#), indexers can automatically pick up where they left off on each run.

Once an indexer is on a schedule, it remains on the schedule until you clear the interval or start time, or set `disabled` to true. Leaving the indexer on a schedule when there's nothing to process won't impact system performance. Checking for changed content is a relatively fast operation.

Prerequisites

- A valid indexer configured with a data source and index.
- [Change detection](#) in the data source. Azure Storage and SharePoint have built-in change detection. Other data sources, such as [Azure SQL](#) and [Azure Cosmos DB](#) must be enabled manually.

Schedule definition

A schedule is part of the indexer definition. If the `schedule` property is omitted, the indexer will only run on demand. The property has two parts.

[] [Expand table](#)

Property	Description
"interval"	(required) The amount of time between the start of two consecutive indexer executions. The smallest interval allowed is 5 minutes, and the longest is 1440 minutes (24 hours). It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601

Property	Description
	<p>duration ↗ value).</p> <p>The pattern for this is: <code>P(nD)(T(nH)(nM))</code>.</p> <p>Examples: <code>PT15M</code> for every 15 minutes, <code>PT2H</code> for every two hours.</p>
"startTime"	(optional) Start time is specified in coordinated universal time (UTC). If omitted, the current time is used. This time can be in the past, in which case the first execution is scheduled as if the indexer has been running continuously since the original start time.

The following example is a schedule that starts on January 1 at midnight and runs every two hours.

JSON

```
{
  "dataSourceName" : "hotels-ds",
  "targetIndexName" : "hotels-idx",
  "schedule" : { "interval" : "PT2H", "startTime" : "2024-01-01T00:00:00Z" }
}
```

Configure a schedule

Schedules are specified in an indexer definition. To set up a schedule, you can use Azure portal, REST APIs, or an Azure SDK.

Azure portal

1. Sign in to the [Azure portal](#) [↗](#) and open the search service page.
2. On the left pane, select **Indexes**.
3. Open an indexer.
4. Select **Settings**.
5. Scroll down to **Schedule**, and then choose Hourly, Daily, or Custom to set a specific date, time, or custom interval.

Switch to the **Indexer Definition (JSON)** tab at the top of the index to view the schedule definition in XSD format.

Scheduling behavior FAQ

Can I run multiple indexer jobs in parallel?

You can run multiple indexers simultaneously, but each indexer is single instance. You can't run two copies of the same indexer concurrently.

For text-based indexing, the scheduler can kick off as many indexer jobs as the search service supports, which is determined by the number of [search units](#). For example, if the service has three replicas and four partitions, you can have 12 indexer jobs in active execution, whether initiated on demand or on a schedule.

For skills-based indexing, indexers run in a specific [execution environment](#). For this reason, the number of service units has no bearing on the number of skills-based indexer jobs you can run. Multiple skills-based indexers can run in parallel, but doing so depends on content processor availability within the execution environment.

Do scheduled jobs always start at the designated time?

Indexer processes can be queued up and might not start exactly at the time posted, depending on the processing workload and other factors. For example, if an indexer happens to still be running when its next scheduled execution is set to start, the pending execution is postponed until the next scheduled occurrence, allowing the current job to finish.

Let's consider an example to make this more concrete. Suppose we configure an indexer schedule with an interval of hourly and a start time of January 1, 2024 at 8:00:00 AM UTC. Here's what could happen when an indexer run takes longer than an hour:

1. The first indexer execution starts at or around January 1, 2024 at 8:00 AM UTC. Assume this execution takes 20 minutes (or any amount of time that's less than 1 hour).
2. The second execution starts at or around January 1, 2024 9:00 AM UTC. Suppose that this execution takes 70 minutes – more than an hour – and it will not complete until 10:10 AM UTC.
3. The third execution is scheduled to start at 10:00 AM UTC, but at that time the previous execution is still running. This scheduled execution is then skipped. The next execution of the indexer won't start until 11:00 AM UTC.

In rare cases, such as during maintenance or when recovering from transient conditions, multiple indexer runs are queued up. When this occurs, the indexer executes pending workloads sequentially within the scheduled window. For example, if an indexer is scheduled to run hourly and several runs were delayed or triggered on-demand, those queued up jobs will execute back-to-back until the queue is drained. These are not additional runs, but represent previously scheduled or requested executions. While this behavior is uncommon in most scenarios, the indexer is designed to eventually process all queued tasks to maintain consistency and data freshness.

Note

If you have strict indexer execution requirements that are time-sensitive, you should consider using the [push API model](#) so you can control the indexing pipeline directly.

What happens if indexing fails repeatedly on the same document?

If an indexer is set to a certain schedule but repeatedly fails on the same document each time, the indexer will begin running on a less frequent interval (up to the maximum interval of at least once every 2 hours or 24 hours, depending on different implementation factors) until it successfully makes progress again. If you believe you have fixed the underlying issue, you can [run the indexer manually](#), and if indexing succeeds, the indexer will return to its regular schedule.

Next steps

For indexers that run on a schedule, you can monitor operations by retrieving status from the search service, or obtain detailed information by enabling resource logging.

- [Monitor search indexer status](#)
- [Collect and analyze log data](#)
- [Index large data sets](#)

Field mappings and transformations using Azure AI Search indexers

09/23/2025



This article explains how to set explicit field mappings that establish the data path between source fields in a supported data source and target fields in a search index.

When to set a field mapping

When an [Azure AI Search indexer](#) loads a search index, it determines the data path using source-to-destination field mappings. Implicit field mappings are internal and occur when field names and data types are compatible between the source and destination. If inputs and outputs don't match, you can define explicit *field mappings* to set up the data path, as described in this article.

Field mappings can also be used for light-weight data conversions, such as encoding or decoding, through [mapping functions](#). If more processing is required, consider [Azure Data Factory](#) to bridge the gap.

Field mappings apply to:

- Physical data structures on both sides of the data path. Logical data structures created by skills reside only in memory. Use [outputFieldMappings](#) to map in-memory nodes to output fields in a search index.
- Parent AI Search indexes only. For "secondary" indexes with "child" documents or "chunks", refer to the [advanced field mapping scenarios](#).
- Top-level search fields only, where the `targetFieldName` is either a simple field or a collection. A target field can't be a complex type.

Supported scenarios

Make sure you're using a [supported data source](#) for indexer-driving indexing.

Use-case	Description
Name discrepancy	<p>Suppose your data source has a field named <code>_city</code>. Given that Azure AI Search doesn't allow field names that start with an underscore, a field mapping lets you effectively map "<code>_city</code>" to "city".</p> <p>If your indexing requirements include retrieving content from multiple data sources, where field names vary among the sources, you could use a field mapping to clarify the path.</p>
Type discrepancy	<p>Supposed you want a source integer field to be of type <code>Edm.String</code> so that it's searchable in the search index. Because the types are different, you'll need to define a field mapping in order for the data path to succeed. Note that Azure AI Search has a smaller set of supported data types than many data sources. If you're importing SQL data, a field mapping allows you to map the SQL data type you want in a search index.</p>
One-to-many data paths	<p>You can populate multiple fields in the index with content from the same source field. For example, you might want to apply different analyzers to each field to support different use cases in your client app.</p>
Encoding and decoding	<p>You can apply mapping functions to support Base64 encoding or decoding of data during indexing.</p>
Split strings or recast arrays into collections	<p>You can apply mapping functions to split a string that includes a delimiter, or to send a JSON array to a search field of type <code>Collection(Edm.String)</code>.</p>

Note

If no field mappings are present, indexers assume data source fields should be mapped to index fields with the same name. Adding a field mapping overrides the default field mappings for the source and target field. Some indexers, such as the [blob storage indexer](#), add default field mappings for the index key field automatically.

Complex fields aren't supported in a field mapping. Your source structure (nested or hierarchical structures) must exactly match the complex type in the index so that the default mappings work. For more information, see [Tutorial: Index nested JSON blobs](#) for an example. If you get an error similar to `"Field mapping specifies target field 'Address/city' that doesn't exist in the index"`, it's because target field mappings can't be a complex type.

Optionally, you might want just a few nodes in the complex structure. To get individual nodes, you can flatten incoming data into a string collection (see [outputFieldMappings](#) for this workaround).

Define a field mapping

This section explains the steps for setting up field mappings.

REST APIs

1. Use [Create Indexer](#) or [Create or Update Indexer](#) or an equivalent method in an Azure SDK. Here's an example of an indexer definition.

JSON

```
{  
  "name": "myindexer",  
  "description": null,  
  "dataSourceName": "mydatasource",  
  "targetIndexName": "myindex",  
  "schedule": { },  
  "parameters": { },  
  "fieldMappings": [],  
  "disabled": false,  
  "encryptionKey": { }  
}
```

2. Fill out the `fieldMappings` array to specify the mappings. A field mapping consists of three parts.

JSON

```
"fieldMappings": [  
  {  
    "sourceFieldName": "_city",  
    "targetFieldName": "city",  
    "mappingFunction": null  

```

 [Expand table](#)

Property	Description
sourceFieldName	Required. Represents a field in your data source.
targetFieldName	Optional. Represents a field in your search index. If omitted, the value of <code>sourceFieldName</code> is assumed for the target. Target fields must be top-level simple fields or collections. It can't be a complex type or collection. If you're handling a data type issue, a field's data type is specified in the index definition. The field mapping just needs to have the field's name.
mappingFunction	Optional. Consists of predefined functions that transform data.

Example: Name or type discrepancy

An explicit field mapping establishes a data path for cases where name and type aren't identical.

Azure AI Search uses case-insensitive comparison to resolve the field and function names in field mappings. This is convenient (you don't have to get all the casing right), but it means that your data source or index can't have fields that differ only by case.

JSON

```
PUT https://[service name].search.windows.net/indexers/myindexer?api-version=[api-version]
Content-Type: application/json
api-key: [admin key]
{
    "dataSourceName" : "mydatasource",
    "targetIndexName" : "myindex",
    "fieldMappings" : [ { "sourceFieldName" : "_city", "targetFieldName" :
"city" } ]
}
```

Example: One-to-many or forked data paths

This example maps a single source field to multiple target fields ("one-to-many" mappings). You can "fork" a field, copying the same source field content to two different index fields that will be analyzed or attributed differently in the index.

JSON

```
"fieldMappings" : [
    { "sourceFieldName" : "text", "targetFieldName" :
"textStandardEnglishAnalyzer" },
    { "sourceFieldName" : "text", "targetFieldName" : "textSoundexAnalyzer" }
]
```

You can use a similar approach for [skills-generated content](#).

Mapping functions and examples

A field mapping function transforms the contents of a field before it's stored in the index. The following mapping functions are currently supported:

- [base64Encode](#)

- [base64Decode](#)
- [extractTokenAtPosition](#)
- [fixedLengthEncode](#)
- [jsonArrayToStringCollection](#)
- [toJson](#)
- [urlEncode](#)
- [urlDecode](#)

Note that these functions are exclusively supported for parent indexes at this time. They aren't compatible with chunked index mapping, therefore, these functions can't be used for [index projections](#).

base64Encode function

Performs *URL-safe* Base64 encoding of the input string. Assumes that the input is UTF-8 encoded.

Example: Base-encoding a document key

Only URL-safe characters can appear in an Azure AI Search document key (so that you can address the document using the [Lookup API](#)). If the source field for your key contains URL-unsafe characters, such as `-` and `\`, use the `base64Encode` function to convert it at indexing time.

The following example specifies the `base64Encode` function on `metadata_storage_name` to handle unsupported characters.

HTTP

```
PUT /indexers?api-version=2025-09-01
{
  "dataSourceName" : "my-blob-datasource",
  "targetIndexName" : "my-search-index",
  "fieldMappings" : [
    {
      "sourceFieldName" : "metadata_storage_name",
      "targetFieldName" : "key",
      "mappingFunction" : {
        "name" : "base64Encode",
        "parameters" : { "useHttpServerUtilityUrlTokenEncode" : false }
      }
    }
  ]
}
```

A document key (both before and after conversion) can't be longer than 1,024 characters.

When you retrieve the encoded key at search time, use the `base64Decode` function to get the original key value, and use that to retrieve the source document.

Example: Make a base-encoded field "searchable"

There are times when you need to use an encoded version of a field like

`metadata_storage_path` as the key, but also need an unencoded version for full text search. To support both scenarios, you can map `metadata_storage_path` to two fields: one for the key (encoded), and a second for a path field that we can assume is attributed as `searchable` in the index schema.

HTTP

```
PUT /indexers/blob-indexer?api-version=2025-09-01
{
    "dataSourceName" : "blob-datasource",
    "targetIndexName" : "my-target-index",
    "schedule" : { "interval" : "PT2H" },
    "fieldMappings" : [
        { "sourceFieldName" : "metadata_storage_path", "targetFieldName" : "key",
        "mappingFunction" : { "name" : "base64Encode" } },
        { "sourceFieldName" : "metadata_storage_path", "targetFieldName" : "path"
    }
]
}
```

Example - preserve original values

The [blob storage indexer](#) automatically adds a field mapping from `metadata_storage_path`, the URI of the blob, to the index key field if no field mapping is specified. This value is Base64 encoded so it's safe to use as an Azure AI Search document key. The following example shows how to simultaneously map a *URL-safe* Base64 encoded version of `metadata_storage_path` to a `index_key` field and preserve the original value in a `metadata_storage_path` field:

JSON

```
"fieldMappings": [
    {
        "sourceFieldName": "metadata_storage_path",
        "targetFieldName": "metadata_storage_path"
    },
    {
        "sourceFieldName": "metadata_storage_path",
        "targetFieldName": "index_key",
    }
]
```

```
        "mappingFunction": {
            "name": "base64Encode"
        }
    }
]
```

If you don't include a parameters property for your mapping function, it defaults to the value `{"useHttpServerUtilityUrlTokenEncode" : true}`.

Azure AI Search supports two different Base64 encodings. You should use the same parameters when encoding and decoding the same field. For more information, see [base64 encoding options](#) to decide which parameters to use.

base64Decode function

Performs Base64 decoding of the input string. The input is assumed to be a *URL-safe* Base64-encoded string.

Example - decode blob metadata or URLs

Your source data might contain Base64-encoded strings, such as blob metadata strings or web URLs, that you want to make searchable as plain text. You can use the `base64Decode` function to turn the encoded data back into regular strings when populating your search index.

JSON

```
"fieldMappings" : [
    {
        "sourceFieldName" : "Base64EncodedMetadata",
        "targetFieldName" : "SearchableMetadata",
        "mappingFunction" : {
            "name" : "base64Decode",
            "parameters" : { "useHttpServerUtilityUrlTokenDecode" : false }
        }
    }
]
```

If you don't include a parameters property, it defaults to the value `{"useHttpServerUtilityUrlTokenEncode" : true}`.

Azure AI Search supports two different Base64 encodings. You should use the same parameters when encoding and decoding the same field. For more information, see [base64 encoding options](#) to decide which parameters to use.

base64 encoding options

Azure AI Search supports URL-safe base64 encoding and normal base64 encoding. A string that is base64 encoded during indexing should be decoded later with the same encoding options, or else the result won't match the original.

If the `useHttpServerUtilityUrlTokenEncode` or `useHttpServerUtilityUrlTokenDecode` parameters for encoding and decoding respectively are set to `true`, then `base64Encode` behaves like `HttpServerUtility.UrlTokenEncode` and `base64Decode` behaves like `HttpServerUtility.UrlTokenDecode`.

⚠ Warning

If `base64Encode` is used to produce key values, `useHttpServerUtilityUrlTokenEncode` must be set to true. Only URL-safe base64 encoding can be used for key values. See [Naming rules](#) for the full set of restrictions on characters in key values.

The .NET libraries in Azure AI Search assume the full .NET Framework, which provides built-in encoding. The `useHttpServerUtilityUrlTokenEncode` and `useHttpServerUtilityUrlTokenDecode` options apply this built-in functionality. If you're using .NET Core or another framework, we recommend setting those options to `false` and calling your framework's encoding and decoding functions directly.

The following table compares different base64 encodings of the string `00>00?00`. To determine the required processing (if any) for your base64 functions, apply your library encode function on the string `00>00?00` and compare the output with the expected output `MDA-MDA_MDA`.

 [Expand table](#)

Encoding	Base64 encode output	Extra processing after library encoding	Extra processing before library decoding
Base64 with padding	MDA+MDA/MDA=	Use URL-safe characters and remove padding	Use standard base64 characters and add padding
Base64 without padding	MDA+MDA/MDA	Use URL-safe characters	Use standard base64 characters
URL-safe base64 with padding	MDA-MDA_MDA=	Remove padding	Add padding
URL-safe base64 without padding	MDA-MDA_MDA	None	None

extractTokenAtPosition function

Splits a string field using the specified delimiter, and picks the token at the specified position in the resulting split.

This function uses the following parameters:

- `delimiter`: a string to use as the separator when splitting the input string.
- `position`: an integer zero-based position of the token to pick after the input string is split.

For example, if the input is `Jane Doe`, the `delimiter` is `" "`(space) and the `position` is 0, the result is `Jane`; if the `position` is 1, the result is `Doe`. If the position refers to a token that doesn't exist, an error is returned.

Example - extract a name

Your data source contains a `PersonName` field, and you want to index it as two separate `FirstName` and `LastName` fields. You can use this function to split the input using the space character as the delimiter.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "PersonName",
    "targetFieldName" : "FirstName",
    "mappingFunction" : { "name" : "extractTokenAtPosition", "parameters" : {
      "delimiter" : " ", "position" : 0 } }
  },
  {
    "sourceFieldName" : "PersonName",
    "targetFieldName" : "LastName",
    "mappingFunction" : { "name" : "extractTokenAtPosition", "parameters" : {
      "delimiter" : " ", "position" : 1 } }
  }
]
```

jsonArrayToStringCollection function

Transforms a string formatted as a JSON array of strings into a string array that can be used to populate a `Collection(Edm.String)` field in the index.

For example, if the input string is `["red", "white", "blue"]`, then the target field of type `Collection(Edm.String)` will be populated with the three values `red`, `white`, and `blue`. For

input values that can't be parsed as JSON string arrays, an error is returned.

Example - populate collection from relational data

Azure SQL Database doesn't have a built-in data type that naturally maps to `Collection(Edm.String)` fields in Azure AI Search. To populate string collection fields, you can preprocess your source data as a JSON string array and then use the `jsonArrayToStringCollection` mapping function.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "tags",
    "mappingFunction" : { "name" : "jsonArrayToStringCollection" }
  }
]
```

urlEncode function

This function can be used to encode a string so that it is "URL safe". When used with a string that contains characters that aren't allowed in a URL, this function will convert those "unsafe" characters into character-entity equivalents. This function uses the UTF-8 encoding format.

Example - document key lookup

`urlEncode` function can be used as an alternative to the `base64Encode` function, if only URL unsafe characters are to be converted, while keeping other characters as-is.

Say, the input string is `<hello>` - then the target field of type `(Edm.String)` will be populated with the value `%3chello%3e`

When you retrieve the encoded key at search time, you can then use the `urlDecode` function to get the original key value, and use that to retrieve the source document.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "SourceKey",
    "targetFieldName" : "IndexKey",
    "mappingFunction" : {
      "name" : "urlEncode"
    }
}
```

```
    }  
]
```

urlDecode function

This function converts a URL-encoded string into a decoded string using UTF-8 encoding format.

Example - decode blob metadata

Some Azure storage clients automatically URL-encode blob metadata if it contains non-ASCII characters. However, if you want to make such metadata searchable (as plain text), you can use the `urlDecode` function to turn the encoded data back into regular strings when populating your search index.

JSON

```
"fieldMappings" : [  
  {  
    "sourceFieldName" : "UrlEncodedMetadata",  
    "targetFieldName" : "SearchableMetadata",  
    "mappingFunction" : {  
      "name" : "urlDecode"  
    }  
  }  
]
```

fixedLengthEncode function

This function converts a string of any length to a fixed-length string.

Example - map document keys that are too long

When errors occur that are related to document key length exceeding 1024 characters, this function can be applied to reduce the length of the document key.

JSON

```
"fieldMappings" : [  
  {  
    "sourceFieldName" : "metadata_storage_path",  
    "targetFieldName" : "your key field",  
    "mappingFunction" : {
```

```
        "name" : "fixedLengthEncode"
    }
]
```

toJson function

This function converts a string into a formatted JSON object. This can be used for scenarios where the data source, such as Azure SQL, doesn't natively support compound or hierarchical data types, and then map it to complex fields.

Example - map text content to a complex field

Assume there's a SQL row with a JSON string that needs to be mapped to a (correspondingly defined) complex field in the index, the `toJson` function can be used to achieve this. For instance, if a complex field in the index needs to be populated with the following data:

JSON

```
{
  "id": "5",
  "info": {
    "name": "Jane",
    "surname": "Smith",
    "skills": [
      "SQL",
      "C#",
      "Azure"
    ],
    "dob": "2005-11-04T12:00:00"
  }
}
```

It can be achieved by using the `toJson` mapping function on a JSON string column in a SQL row that looks like this: `{"id": 5, "info": {"name": "Jane", "surname": "Smith", "skills": ["SQL", "C#", "Azure"]}, "dob": "2005-11-04T12:00:00"}`.

The field mapping needs to be specified as shown below.

JSON

```
"fieldMappings" : [
{
  "sourceFieldName" : "content",
  "targetFieldName" : "complexField",
```

```
        "mappingFunction" : {  
            "name" : "toJson"  
        }  
    }  
]
```

Advanced field mapping scenarios

In scenarios where you have "one-to-many" document relationships, such as data chunking or splitting, follow these guidelines for mapping fields from parent documents to "child" documents (chunks):

1. Skipping parent document indexing

If you are skipping the indexing of parent documents (by setting `projectionMode` to `skipIndexingParentDocuments` in the skillset's `indexProjections`), use [index projections](#) to map fields from the parent documents to the "child" documents.

2. Indexing both parent and "child" documents

If you are indexing both parent documents and "child" documents:

- Use field mappings to map fields to the parent documents.
- Use [index projections](#) to map fields to the "child" documents.

3. Mapping function-transformed values to parent and/or "child" documents

If a field in the parent document requires a transformation (using the [mapping functions](#) such as encoding) and needs to be mapped to the parent and/or "child" documents:

- Apply the transformation using field mappings' [functions](#) in the indexer.
- Use [index projections](#) in the skillset to map the transformed field to the "child" documents.

See also

- [Supported data types in Azure AI Search](#)
- [SQL data type map](#)

Content metadata properties used in Azure AI Search

Article • 04/14/2025

Several indexer-supported data sources, including Azure Blob Storage, Azure Data Lake Storage Gen2, and SharePoint, contain standalone files or embedded objects of various content types. Many of those content types have metadata properties that can be useful to index. Just as you can create search fields for standard blob properties like `metadata_storage_name`, you can create fields in a search index for metadata properties that are specific to a document format.

Supported document formats

Azure AI Search supports blob indexing and SharePoint document indexing for the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Document format properties

The following table summarizes processing for each document format, and describes the metadata properties extracted by a blob indexer and the SharePoint Online indexer.

 Expand table

Document format / content type	Extracted metadata	Processing details
CSV (text/csv)	<code>metadata_content_type</code> <code>metadata_content_encoding</code>	Extract text NOTE: If you need to extract multiple document fields from a CSV blob, see Index CSV blobs
DOC (application/msword)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_character_count</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_page_count</code> <code>metadata_word_count</code>	Extract text, including embedded documents
DOCM (application/vnd.ms-word.document.macroenabled.12)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_character_count</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_page_count</code> <code>metadata_word_count</code>	Extract text, including embedded documents
DOCX (application/vnd.openxmlformats-officedocument.wordprocessingml.document)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_character_count</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_page_count</code> <code>metadata_word_count</code>	Extract text, including embedded documents
EML (message/rfc822)	<code>metadata_content_type</code> <code>metadata_message_from</code> <code>metadata_message_to</code> <code>metadata_message_cc</code> <code>metadata_creation_date</code> <code>metadata_subject</code>	Extract text, including attachments
EPUB (application/epub+zip)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_title</code> <code>metadata_description</code> <code>metadata_language</code> <code>metadata_keywords</code> <code>metadata_identifier</code> <code>metadata_publisher</code>	Extract text from all documents in the archive
GZ (application/gzip)	<code>metadata_content_type</code>	Extract text from all

Document format / content type	Extracted metadata	Processing details
		documents in the archive
HTML (text/html or application/xhtml+xml)	<code>metadata_content_encoding</code> <code>metadata_content_type</code> <code>metadata_language</code> <code>metadata_description</code> <code>metadata_keywords</code> <code>metadata_title</code>	Strip HTML elements and extract text
JSON (application/json)	<code>metadata_content_type</code> <code>metadata_content_encoding</code>	Extract text NOTE: If you need to extract multiple document fields from a JSON blob, see Index JSON blobs
KML (application/vnd.google-earth.kml+xml)	<code>metadata_content_type</code> <code>metadata_content_encoding</code> <code>metadata_language</code>	Strip XML elements and extract text
MSG (application/vnd.ms-outlook)	<code>metadata_content_type</code> <code>metadata_message_from</code> <code>metadata_message_from_email</code> <code>metadata_message_to</code> <code>metadata_message_to_email</code> <code>metadata_message_cc</code> <code>metadata_message_cc_email</code> <code>metadata_message_bcc</code> <code>metadata_message_bcc_email</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_subject</code>	Extract text, including text extracted from attachments. <code>metadata_message_to_email</code> , <code>metadata_message_cc_email</code> , and <code>metadata_message_bcc_email</code> are string collections. The rest of the fields are strings.
ODP (application/vnd.oasis.opendocument.presentation)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_title</code>	Extract text, including embedded documents
ODS (application/vnd.oasis.opendocument.spreadsheet)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code>	Extract text, including embedded documents
ODT (application/vnd.oasis.opendocument.text)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_character_count</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code>	Extract text, including embedded documents

Document format / content type	Extracted metadata	Processing details
	metadata_page_count metadata_word_count	
PDF (application/pdf)	metadata_content_type metadata_language metadata_author metadata_title metadata_creation_date	Extract text, including embedded documents (excluding images)
Plain text (text/plain)	metadata_content_type metadata_content_encoding metadata_language	Extract text
PPT (application/vnd.ms-powerpoint)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified metadata_slide_count metadata_title	Extract text, including embedded documents
PPTM (application/vnd.ms-powerpoint.presentation.macroenabled.12)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified metadata_slide_count metadata_title	Extract text, including embedded documents
PPTX (application/vnd.openxmlformats-officedocument.presentationml.presentation)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified metadata_slide_count metadata_title	Extract text, including embedded documents
RTF (application/rtf)	metadata_content_type metadata_author metadata_character_count metadata_creation_date metadata_last_modified metadata_page_count metadata_word_count	Extract text
WORD 2003 XML (application/vnd.ms-wordml)	metadata_content_type metadata_author metadata_creation_date	Strip XML elements and extract text
WORD XML (application/vnd.ms-word2006ml)	metadata_content_type metadata_author metadata_character_count	Strip XML elements and extract text

Document format / content type	Extracted metadata	Processing details
	metadata_creation_date metadata_last_modified metadata_page_count metadata_word_count	
XLS (application/vnd.ms-excel)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified	Extract text, including embedded documents
XLSM (application/vnd.ms-excel.sheet.macroenabled.12)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified	Extract text, including embedded documents
XLSX (application/vnd.openxmlformats-officedocument.spreadsheetml.sheet)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified	Extract text, including embedded documents
XML (application/xml)	metadata_content_type metadata_content_encoding metadata_language	Strip XML elements and extract text
ZIP (application/zip)	metadata_content_type	Extract text from all documents in the archive

Related content

- [Indexers in Azure AI Search](#)
- [AI enrichment in Azure AI Search](#)
- [Search over Azure Blob Storage content](#)
- [Index data from SharePoint](#)

Indexing blobs and files to produce multiple search documents

10/09/2025

Applies to: [Blob indexers](#), [File indexers](#)

By default, an indexer treats the contents of a blob or file as a single search document. If you want a more granular representation in a search index, you can set **parsingMode** values to create multiple search documents from one blob or file. The **parsingMode** values that result in many search documents include `delimitedText` (for [CSV](#)), and `jsonArray` or `jsonLines` (for [JSON](#)).

When you use any of these parsing modes, the new search documents that emerge must have unique document keys, and a problem arises in determining where that value comes from. The parent blob has at least one unique value in the form of `metadata_storage_path` property, but if it contributes that value to more than one search document, the key is no longer unique in the index.

To address this problem, the blob indexer generates an `AzureSearch_DocumentKey` that uniquely identifies each child search document created from the single blob parent. This article explains how this feature works.

One-to-many document key

A document key uniquely identifies each document in an index. When no parsing mode is specified, and if there's no [explicit field mapping](#) in the indexer definition for the search document key, the blob indexer automatically maps the `metadata_storage_path` property as the document key. This default mapping ensures that each blob appears as a distinct search document. It also eliminates the need for you to manually create this field mapping. Normally, fields with identical names and types are the only ones mapped automatically.

In a one-to-many search document scenario, an implicit document key based on `metadata_storage_path` property isn't possible. As a workaround, Azure AI Search can generate a document key for each individual entity extracted from a blob. The system generates a key called `AzureSearch_DocumentKey` and adds it to each search document. The indexer keeps track of the "many documents" created from each blob, and can target updates to the search index when source data changes over time.

By default, when no explicit field mappings for the key index field are specified, the `AzureSearch_DocumentKey` is mapped to it, using the `base64Encode` field-mapping function.

Example

Assume an index definition with the following fields:

- `id`
- `temperature`
- `pressure`
- `timestamp`

And your blob container has blobs with the following structure:

Blob1.json

JSON

```
{ "temperature": 100, "pressure": 100, "timestamp": "2024-02-13T00:00:00Z" }  
{ "temperature" : 33, "pressure" : 30, "timestamp": "2024-02-14T00:00:00Z" }
```

Blob2.json

JSON

```
{ "temperature": 1, "pressure": 1, "timestamp": "2023-01-12T00:00:00Z" }  
{ "temperature" : 120, "pressure" : 3, "timestamp": "2022-05-11T00:00:00Z" }
```

When you create an indexer and set the `parsingMode` to `jsonLines` - without specifying any explicit field mappings for the key field, the following mapping is applied implicitly.

HTTP

```
{  
  "sourceFieldName" : "AzureSearch_DocumentKey",  
  "targetFieldName": "id",  
  "mappingFunction": { "name" : "base64Encode" }  
}
```

This setup results in disambiguated document keys, similar to the following illustration (base64-encoded ID shortened for brevity).

[] Expand table

ID	temperature	pressure	timestamp
aHR0 ... YjEuanNvbjsx	100	100	2024-02-13T00:00:00Z

ID	temperature	pressure	timestamp
aHR0 ... YjEuanNvbjsy	33	30	2024-02-14T00:00:00Z
aHR0 ... YjluanNvbjsx	1	1	2023-01-12T00:00:00Z
aHR0 ... YjluanNvbjsy	120	3	2022-05-11T00:00:00Z

Custom field mapping for index key field

Assuming the same index definition as the previous example, suppose your blob container has blobs with the following structure:

Blob1.json

JSON

```
recordid, temperature, pressure, timestamp
1, 100, 100, "2024-02-13T00:00:00Z"
2, 33, 30, "2024-02-14T00:00:00Z"
```

Blob2.json

JSON

```
recordid, temperature, pressure, timestamp
1, 1, 1, "20123-01-12T00:00:00Z"
2, 120, 3, "2022-05-11T00:00:00Z"
```

When you create an indexer with `delimitedText` **parsingMode**, it might feel natural to set up a field-mapping function to the key field as follows:

HTTP

```
{
  "sourceFieldName" : "recordid",
  "targetFieldName": "id"
}
```

However, this mapping doesn't result in four documents showing up in the index because the `recordid` field isn't unique *across blobs*. Hence, we recommend you to make use of the implicit field mapping applied from the `AzureSearch_DocumentKey` property to the key index field for "one-to-many" parsing modes.

If you do want to set up an explicit field mapping, make sure that the `sourceField` is distinct for each individual entity **across all blobs**.

! Note

The approach used by `AzureSearch_DocumentKey` of ensuring uniqueness per extracted entity is subject to change and therefore you shouldn't rely on its value for your application's needs.

Specify the index key field in your data

Assuming the same index definition as the previous example and `parsingMode` is set to `jsonLines` without specifying any explicit field mappings so the mappings look like in the first example, suppose your blob container has blobs with the following structure:

Blob1.json

JSON

```
id, temperature, pressure, timestamp
1, 100, 100, "2024-02-13T00:00:00Z"
2, 33, 30, "2024-02-14T00:00:00Z"
```

Blob2.json

JSON

```
id, temperature, pressure, timestamp
1, 1, 1, "2023-01-12T00:00:00Z"
2, 120, 3, "2022-05-11T00:00:00Z"
```

Each document contains the `id` field, which is defined as the `key` field in the index. In this situation, the system generates a unique `AzureSearch_DocumentKey` for the document, but it isn't used as the "key." Instead, the value of the `id` field is mapped to the `key`` field.

Similar to the previous example, this mapping doesn't result in four documents showing up in the index because the `id` field isn't unique *across blobs*. When this situation occurs, any JSON entry that specifies an `id` causes a merge with the existing document instead of uploading a new one. The index then reflects the latest state of the entry with the specified `id`.

Limitations

When a document entry in the index is created from a line in a file, as explained in this article, deleting that line from the file doesn't automatically remove the corresponding entry from the index. To delete the document entry, you must manually submit a deletion request to the index using the [REST API deletion operation](#).

Next steps

If you aren't already familiar with the basic structure and workflow of blob indexing, you should review [Indexing Azure Blob Storage with Azure AI Search](#) first. For more information about parsing modes for different blob content types, review the following articles.

[Indexing CSV blobs](#)

[Indexing JSON blobs](#)

Index plain text blobs and files in Azure AI Search

10/09/2025

Applies to: [Blob indexers](#), [File indexers](#)

When using an indexer to extract searchable blob text or file content for full text search, you can assign a parsing mode to get better indexing outcomes. By default, the indexer parses a blob's `content` property as a single chunk of text. However, if all blobs and files contain plain text in the same encoding, you can significantly improve indexing performance by using the `text` parsing mode.

Recommendations for `text` parsing include either of the following characteristics:

- File type is `.txt`
- Files are of any type, but the content itself is text (for example, program source code, HTML, XML, and so forth). For files in a markup language, the syntax characters come through as static text.

Recall that all indexers serialize to JSON. By default, the content of the entire text file is indexed within one large field as `"content": "<file-contents>"`. New line and return instructions are embedded in the content field and expressed as `\r\n\`.

If you want a more refined or granular outcome, and if the file type is compatible, consider the following solutions:

- [delimitedText](#) parsing mode, if the source is CSV
- [jsonArray](#) or [jsonLines](#), if the source is JSON

An alternative third option for breaking content into multiple parts requires advanced features in the form of [AI enrichment](#). It adds analysis that identifies and assigns chunks of the file to different search fields. You might find a full or partial solution through [built-in skills](#) such as entity recognition or keyword extraction, but a more likely solution might be a custom learning model that understands your content, wrapped in a [custom skill](#).

Set up plain text indexing

To index plain text blobs, create or update an indexer definition with the `parsingMode` configuration property set to `text` on a [Create Indexer](#) request:

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "parsingMode" : "text" } }
}
```

By default, the `UTF-8` encoding is assumed. To specify a different encoding, use the `encoding` configuration property. The supported [list of encodings](#) is under [.NET 5 and later support](#) column.

HTTP

```
{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "parsingMode" : "text", "encoding" : "iso-8859-1" } }
}
```

Request example

Parsing modes are specified in the indexer definition.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-plaintext-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "parameters" : { "configuration" : { "parsingMode" : "delimitedText",
    "delimitedTextHeaders" : "id,datePublished,tags" } }
}
```

Next steps

- [Indexers in Azure AI Search](#)
- [How to configure a blob indexer](#)
- [Blob indexing overview](#)

Change and delete detection using indexers for Azure Storage in Azure AI Search

10/09/2025

After an initial search index is created, you might want subsequent indexer jobs to only pick up new and changed documents. For indexed content that originates from Azure Storage, change detection occurs automatically because indexers keep track of the last update using the built-in timestamps on objects and files in Azure Storage.

Although change detection is a given, deletion detection isn't. An indexer doesn't track object deletion in data sources. To avoid having orphan search documents, you can implement a "soft delete" strategy that results in deleting search documents first, with physical deletion in Azure Storage following as a second step.

There are two ways to implement a soft delete strategy:

- [Native blob soft delete](#), applies to Blob Storage only
- [Soft delete using custom metadata](#)

The deletion detection strategy must be applied from the very first indexer run. If you didn't establish the deletion policy prior to the initial run, any documents that were deleted before the policy was implemented will remain in your index, even if you add the policy to the indexer later and reset it. If this has occurred, it's suggested that you create a new index using a new indexer, ensuring the deletion policy is in place from the beginning.

Prerequisites

- Use an Azure Storage indexer for [Blob Storage](#), [Table Storage](#), [File Storage](#), or [Data Lake Storage Gen2](#)
- Use consistent document keys and file structure. Changing document keys or directory names and paths (applies to ADLS Gen2) breaks the internal tracking information used by indexers to know which content was indexed, and when it was last indexed.

Note

ADLS Gen2 allows directories to be renamed. When a directory is renamed, the timestamps for the blobs in that directory don't get updated. As a result, the indexer won't reindex those blobs. If you need the blobs in a directory to be reindexed after a directory

rename because they now have new URLs, you need to update the `LastModified` timestamp for all the blobs in the directory so that the indexer knows to reindex them during a future run. The virtual directories in Azure Blob Storage can't be changed, so they don't have this issue.

Native blob soft delete

For this deletion detection approach, Azure AI Search depends on the [native blob soft delete](#) feature in Azure Blob Storage to determine whether blobs have transitioned to a soft deleted state. When blobs are detected in this state, a search indexer uses this information to remove the corresponding document from the index.

Requirements for native soft delete

- Blobs must be in an Azure Blob Storage container, including ADLS Gen2 Blob container. The Azure AI Search native blob soft delete policy isn't supported for Azure Files.
- [Enable soft delete for blobs](#).
- Document keys for the documents in your index must be mapped to either be a blob property or blob metadata, such as "metadata_storage_path".
- You can use the [REST API](#), or the indexer Data Source configuration in the Azure portal, to configure support for soft delete.
- [Blob versioning](#) must not be enabled in the storage account. Otherwise, native soft delete isn't supported by design.

Configure native soft delete

In Blob storage, when enabling soft delete per the requirements, set the retention policy to a value that's much higher than your indexer interval schedule. If there's an issue running the indexer, or if you have a large number of documents to index, there's plenty of time for the indexer to eventually process the soft deleted blobs. Azure AI Search indexers will only delete a document from the index if it processes the blob while it's in a soft deleted state.

In Azure AI Search, set a native blob soft deletion detection policy on the data source. You can do this either from the Azure portal or by using the [REST API](#). The following instructions explain how to set the delete detection policy in Azure portal or through REST APIs.

Azure portal

1. Sign in to the [Azure portal](#).
2. On the Azure AI Search service Overview page, go to **New Data Source**, a visual editor for specifying a data source definition.

The following screenshot shows where you can find this feature in the Azure portal.

The screenshot shows the Azure AI Search service Overview page for a search service named "azure-cognitive-search-service". The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Settings. The main content area has tabs for Get Started, Usage, Monitoring, Indexes, Indexers, and Data sources. The "Data sources" tab is selected, and a red box highlights the "+ New Data Source" button. Below it, there is a table with one row labeled "my-blob-data-source".

3. On the **New Data Source** form, fill out the required fields, select the **Track deletions** checkbox and choose **Native blob soft delete**. Then hit **Save** to enable the feature on Data Source creation.

The screenshot shows the "New Data Source" form for creating a data source. The "Settings" tab is selected. The "Data Source" dropdown is set to "Azure Blob Storage". The "Name" field is filled with "my-blob-data-source". The "Connection string" field contains "DefaultEndpointsProtocol=https;AccountName=[accountName];AccountKey=[accountKey]". Under "Managed identity authentication", the "None" option is selected. The "Container name" field is set to "mycontainer". The "Blob folder" field is set to "your/folder/here". The "Track deletions" checkbox is checked. Under "Data deletion detection policy", the "Native blob soft delete" radio button is selected. A red box highlights the "Native blob soft delete" radio button. The "Description" field is empty and labeled "(optional)".

Reindex undeleted blobs using native soft delete policies

If you restore a soft deleted blob in Blob storage, the indexer won't always reindex it. This is because the indexer uses the blob's `LastModified` timestamp to determine whether indexing is needed. When a soft deleted blob is undeleted, its `LastModified` timestamp doesn't get updated, so if the indexer has already processed blobs with more recent `LastModified` timestamps, it won't reindex the undeleted blob.

To make sure that an undeleted blob is reindexed, you'll need to update the blob's `LastModified` timestamp. One way to do this is by resaving the metadata of that blob. You don't need to change the metadata, but resaving the metadata will update the blob's `LastModified` timestamp so that the indexer knows to pick it up.

Soft delete strategy using custom metadata

This method uses custom metadata to indicate whether a search document should be removed from the index. It requires two separate actions: deleting the search document from the index, followed by file deletion in Azure Storage.

This feature is generally available.

There are steps to follow in both Azure Storage and Azure AI Search, but there are no other feature dependencies.

1. In Azure Storage, add a custom metadata key-value pair to the file to indicate the file is flagged for deletion. For example, you could name the property "IsDeleted", set to false. When you want to delete the file, change it to true.
2. In Azure AI Search, edit the data source definition to include a "dataDeletionDetectionPolicy" property. For example, the following policy considers a file to be deleted if it has a metadata property `IsDeleted` with the value `true`:

HTTP

```
PUT https://[service name].search.windows.net/datasources/file-datasource?  
api-version=2025-09-01  
{  
    "name" : "file-datasource",  
    "type" : "azurefile",  
    "credentials" : { "connectionString" : "<your storage connection string>" },  
    "container" : { "name" : "my-share", "query" : null },  
    "dataDeletionDetectionPolicy" : {  
        "@odata.type"  
        :"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
        "softDeleteColumnName" : "IsDeleted",  
        "softDeleteMarkerValue" : "true"
```

```
    }  
}
```

3. Run the indexer. Once the indexer has processed the file and deleted the document from the search index, you can then delete the physical file in Azure Storage.

Reindex undeleted blobs and files

You can reverse a soft-delete if the original source file still physically exists in Azure Storage.

1. Change the `"softDeleteMarkerValue" : "false"` on the blob or file in Azure Storage.
2. Check the blob or file's `LastModified` timestamp to make it's newer than the last indexer run. You can force an update to the current date and time by resaving the existing metadata.
3. Run the indexer.

Next steps

- [Indexers in Azure AI Search](#)
- [How to configure a blob indexer](#)
- [Blob indexing overview](#)

Index JSON blobs and files in Azure AI Search

10/09/2025

Applies to: [Blob indexers](#), [File indexers](#)

For blob indexing in Azure AI Search, this article shows you how to set properties for blobs or files consisting of JSON documents. JSON files in Azure Blob Storage or Azure Files commonly assume any of these forms:

- A single JSON document
- A JSON document containing an array of well-formed JSON elements
- A JSON document containing multiple entities, separated by a newline

The blob indexer provides a `parsingMode` parameter to optimize the output of the search document based on JSON structure. Parsing modes consist of the following options:

[] [Expand table](#)

parsingMode	JSON document	Description
<code>json</code>	One per blob	(default) Parses JSON blobs as a single chunk of text. Each JSON blob becomes a single search document.
<code>jsonArray</code>	Multiple per blob	Parses a JSON array in the blob, where each element of the array becomes a separate search document.
<code>jsonLines</code>	Multiple per blob	Parses a blob that contains multiple JSON entities (also an array), with individual elements separated by a newline. The indexer starts a new search document after each new line.

For both `jsonArray` and `jsonLines`, you should review [Indexing one blob to produce many search documents](#) to understand how the blob indexer handles disambiguation of the document key for multiple search documents produced from the same blob.

Within the indexer definition, you can optionally set [field mappings](#) to choose which properties of the source JSON document are used to populate your target search index. For example, when using the `jsonArray` parsing mode, if the array exists as a lower-level property, you can set a "documentRoot" property indicating where the array is placed within the blob.

! **Note**

When a JSON parsing mode is used, Azure AI Search assumes that all blobs use the same parser (either for `json`, `jsonArray` or `jsonLines`). If you have a mix of different file types in the same data source, consider using [file extension filters](#) to control which files are imported.

The following sections describe each mode in more detail. If you're unfamiliar with indexer clients and concepts, see [Create a search indexer](#). You should also be familiar with the details of [basic blob indexer configuration](#), which isn't repeated here.

Index single JSON documents (one per blob)

By default, blob indexers parse JSON blobs as a single chunk of text, one search document for each blob in a container. If the JSON is structured, the search document can reflect that structure, with individual elements represented as individual fields. For example, assume you have the following JSON document in Azure Blob Storage:

HTTP

```
{  
  "article" : {  
    "text" : "A hopefully useful article explaining how to parse JSON blobs",  
    "datePublished" : "2020-04-13",  
    "tags" : [ "search", "storage", "howto" ]  
  }  
}
```

The blob indexer parses the JSON document into a single search document, loading an index by matching "text", "datePublished", and "tags" from the source against identically named and typed target index fields. Given an index with "text", "datePublished", and "tags" fields, the blob indexer can infer the correct mapping without a field mapping present in the request.

Although the default behavior is one search document per JSON blob, setting the `json` parsing mode changes the internal field mappings for content, promoting fields inside `content` to actual fields in the search index. An example indexer definition for the `json` parsing mode might look like this:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01  
Content-Type: application/json  
api-key: [admin key]  
  
{  
  "name" : "my-json-indexer",
```

```
"dataSourceName" : "my-blob-datasource",
"targetIndexName" : "my-target-index",
"parameters" : { "configuration" : { "parsingMode" : "json" } }
}
```

ⓘ Note

As with all indexers, if fields don't clearly match, you should expect to explicitly specify individual [field mappings](#) unless you're using the implicit fields mappings available for blob content and metadata, as described in [basic blob indexer configuration](#). To override an existing index value, the source JSON must provide a non-null value. If the field in the source document is null, the indexer will retain the existing value. To explicitly clear a field, pass an empty string ("") instead. This prevents unintended deletions from the index.

json example (single hotel JSON files)

The [hotel JSON document data set](#) on GitHub is helpful for testing JSON parsing, where each blob represents a structured JSON file. You can upload the data files to Blob Storage and use an [import wizard](#) to quickly evaluate how this content is parsed into individual search documents.

The data set consists of five blobs, each containing a hotel document with an address collection and a rooms collection. The blob indexer detects both collections and reflects the structure of the input documents in the index schema.

Parse JSON arrays

Alternatively, you can use the JSON array option. This option is useful when blobs contain an array of well-formed JSON objects, and you want each element to become a separate search document. Using `jsonArrays`, the following JSON blob produces three separate documents, each with `"id"` and `"text"` fields.

```
text
```

```
[  
  { "id" : "1", "text" : "example 1" },  
  { "id" : "2", "text" : "example 2" },  
  { "id" : "3", "text" : "example 3" }  
]
```

The `parameters` property on the indexer contains parsing mode values. For a JSON array, the indexer definition should look similar to the following example.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "parameters" : { "configuration" : { "parsingMode" : "jsonArray" } }
}
```

jsonArrays example

The [New York Philharmonic JSON data set](#) on GitHub is helpful for testing JSON array parsing. You can upload the data files to Blob storage and use an [import wizard](#) to quickly evaluate how this content is parsed into individual search documents.

The data set consists of eight blobs, each containing a JSON array of entities, for a total of 100 entities. The entities vary as to which fields are populated, but the end result is one search document per entity, from all arrays, in all blobs.

Parsing nested JSON arrays

For JSON arrays having nested elements, you can specify a `documentRoot` to indicate a multi-level structure. For example, if your blobs look like this:

HTTP

```
{
    "level1" : {
        "level2" : [
            { "id" : "1", "text" : "Use the documentRoot property" },
            { "id" : "2", "text" : "to pluck the array you want to index" },
            { "id" : "3", "text" : "even if it's nested inside the document" }
        ]
    }
}
```

Use this configuration to index the array contained in the `level2` property:

HTTP

```
{  
  "name" : "my-json-array-indexer",  
  ... other indexer properties  
  "parameters" : { "configuration" : { "parsingMode" : "jsonArray",  
"documentRoot" : "/level1/level2" } }  
}
```

Parse JSON entities separated by newlines

If your blob contains multiple JSON entities separated by a newline, and you want each element to become a separate search document, use `jsonLines`.

text

```
{ "id" : "1", "text" : "example 1" }  
{ "id" : "2", "text" : "example 2" }  
{ "id" : "3", "text" : "example 3" }
```

For JSON lines, the indexer definition should look similar to the following example.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01  
Content-Type: application/json  
api-key: [admin key]  
  
{  
  "name" : "my-json-indexer",  
  "dataSourceName" : "my-blob-datasource",  
  "targetIndexName" : "my-target-index",  
  "parameters" : { "configuration" : { "parsingMode" : "jsonLines" } }  
}
```

Map JSON fields to search fields

Field mappings associate a source field with a destination field in situations where the field names and types aren't identical. But field mappings can also be used to match parts of a JSON document and "lift" them into top-level fields of the search document.

The following example illustrates this scenario. For more information about field mappings in general, see [field mappings](#).

HTTP

```
{  
  "article" : {  
    "text" : "A hopefully useful article explaining how to parse JSON blobs",  
    "datePublished" : "2016-04-13"  
    "tags" : [ "search", "storage", "howto" ]  
  }  
}
```

Assume a search index with the following fields: `text` of type `Edm.String`, `date` of type `Edm.DateTimeOffset`, and `tags` of type `Collection(Edm.String)`. Notice the discrepancy between "datePublished" in the source and `date` field in the index. To map your JSON into the desired shape, use the following field mappings:

HTTP

```
"fieldMappings" : [  
  { "sourceFieldName" : "/article/text", "targetFieldName" : "text" },  
  { "sourceFieldName" : "/article/datePublished", "targetFieldName" : "date" },  
  { "sourceFieldName" : "/article/tags", "targetFieldName" : "tags" }  
]
```

Source fields are specified using the [JSON Pointer](#) notation. You start with a forward slash to refer to the root of your JSON document, then pick the desired property (at arbitrary level of nesting) by using forward slash-separated path.

You can also refer to individual array elements by using a zero-based index. For example, to pick the first element of the "tags" array from the above example, use a field mapping like this:

HTTP

```
{ "sourceFieldName" : "/article/tags/0", "targetFieldName" : "firstTag" }
```

ⓘ Note

If "sourceFieldName" refers to a property that doesn't exist in the JSON blob, that mapping is skipped without an error. This behavior allows indexing to continue for JSON blobs that have a different schema (which is a common use case). Because there's no validation check, check the mappings carefully for typos so that you aren't losing documents for the wrong reason.

Next steps

- [Configure blob indexers](#)
- [Define field mappings](#)
- [Indexers overview](#)
- [How to index CSV blobs with a blob indexer](#)
- [Tutorial: Search semi-structured data from Azure Blob Storage](#)

Index Markdown blobs and files in Azure AI Search

10/09/2025

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In Azure AI Search, indexers for Azure Blob Storage, Azure Files, and Microsoft OneLake support a `markdown` parsing mode for Markdown files. Markdown files can be indexed in two ways:

- One-to-many parsing mode, creating multiple search documents per Markdown file
- One-to-one parsing mode, creating one search document per Markdown file

💡 Tip

Continue on to the [Tutorial: Search Markdown data from Azure Blob Storage](#) after reviewing this article.

Prerequisites

- A supported data source: Azure Blob storage, Azure File storage, Microsoft OneLake.

For OneLake, make sure you meet all of the requirements of the [OneLake indexer](#).

Azure Storage for [blob indexers](#) and [file indexers](#) is a standard performance (general-purpose v2) instance that supports hot and cool access tiers.

Markdown parsing mode parameters

Parsing mode parameters are specified in an indexer definition when you create or update an indexer.

HTTP

```

POST https://[service name].search.windows.net/indexers?api-version=2025-08-01-preview
Content-Type: application/json
api-key: [admin key]

{
  "name": "my-markdown-indexer",
  "dataSourceName": "my-blob-datasource",
  "targetIndexName": "my-target-index",
  "parameters": {
    "configuration": {
      "parsingMode": "markdown",
      "markdownParsingSubmode": "oneToMany",
      "markdownHeaderDepth": "h6"
    }
  },
}

```

The blob indexer provides a `submode` parameter to determine the output of structure of the search documents. Markdown parsing mode provides the following submode options:

 Expand table

parsingMode	submode	Search document	Description
<code>markdown</code>	<code>oneToMany</code>	Multiple per blob	(default) Breaks the Markdown into multiple search documents, each representing a content (nonheader) section of the Markdown file. You can omit submode unless you want one-to-one parsing.
<code>markdown</code>	<code>oneToOne</code>	One per blob	Parses the Markdown into one search document, with sections mapped to specific headers in the Markdown file.

For `oneToMany` submode, you should review [Indexing one blob to produce many search documents](#) to understand how the blob indexer handles disambiguation of the document key for multiple search documents produced from the same blob.

Later sections describe each submode in more detail. If you're unfamiliar with indexer clients and concepts, see [Create a search indexer](#). You should also be familiar with the details of [basic blob indexer configuration](#), which isn't repeated here.

Optional Markdown parsing parameters

Parameters are case-sensitive.

Parameter name	Allowed Values	Description
markdownHeaderDepth	h1, h2, h3, h4, h5, h6(default)	This parameter determines the deepest header level that is considered when parsing, allowing for flexible handling of document structure (for example, when <code>markdownHeaderDepth</code> is set to <code>h1</code> , the parser only recognizes top-level headers that begin with "#", and all lower-level headers are treated as plain text). If not specified, it defaults to <code>h6</code> .

This setting can be changed after initial creation of the indexer, however the structure of the resulting search documents might change depending on the Markdown content.

Supported Markdown elements

Markdown parsing only splits content based on headers. All other elements such as lists, code blocks, tables, and so forth, are treated as plain text and passed into a content field.

Sample Markdown content

The following Markdown content is used for the examples on this page:

Markdown
<pre># Section 1 Content for section 1. ## Subsection 1.1 Content for subsection 1.1. # Section 2 Content for section 2.</pre>

Use one-to-many parsing mode

The one-to-many parsing mode parses Markdown files into multiple search documents, where each document corresponds to a specific content section of the Markdown file based on the header metadata at that point in the document. The Markdown is parsed based on headers into search documents, which contain the following content:

- `content`: A string that contains the raw Markdown found in a specific location, based on the header metadata at that point in the document.
- `sections`: An object that contains subfields for the header metadata up to the desired header level. For example, when `markdownHeaderDepth` is set to `h3`, contains string fields `h1`, `h2`, and `h3`. These fields are indexed by mirroring this structure in the index, or through field mappings in the format `/sections/h1`, `sections/h2`, etc. See index and indexer configurations in the following samples for in-context examples. The subfields contained are:
 - `h1` - A string containing the `h1` header value. Empty string if not set at this point in the document.
 - (Optional) `h2` - A string containing the `h2` header value. Empty string if not set at this point in the document.
 - (Optional) `h3` - A string containing the `h3` header value. Empty string if not set at this point in the document.
 - (Optional) `h4` - A string containing the `h4` header value. Empty string if not set at this point in the document.
 - (Optional) `h5` - A string containing the `h5` header value. Empty string if not set at this point in the document.
 - (Optional) `h6` - A string containing the `h6` header value. Empty string if not set at this point in the document.
- `ordinal_position`: An integer value indicating the position of the section within the document hierarchy. This field is used for ordering the sections in their original sequence as they appear in the document, beginning with an ordinal position of 1 and incrementing sequentially for each header.

Index schema for one-to-many parsing

An example index configuration might look something like this:

HTTP

```
{
  "name": "my-markdown-index",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true
    },
    {
      "name": "content",
      "type": "Edm.String",
    }
  ]
}
```

```
},
{
  "name": "ordinal_position",
  "type": "Edm.Int32"
},
{
  "name": "sections",
  "type": "Edm.ComplexType",
  "fields": [
    {
      "name": "h1",
      "type": "Edm.String"
    },
    {
      "name": "h2",
      "type": "Edm.String"
    }
  ]
}
}
```

Indexer definition for one-to-many parsing

If field names and data types align, the blob indexer can infer the mapping without an explicit field mapping present in the request, so an indexer configuration corresponding to the provided index configuration might look like this:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-08-01-preview
Content-Type: application/json
api-key: [admin key]

{
  "name": "my-markdown-indexer",
  "dataSourceName": "my-blob-datasource",
  "targetIndexName": "my-target-index",
  "parameters": {
    "configuration": { "parsingMode": "markdown" }
  },
}
```

⚠ Note

The `submode` doesn't need to be set explicitly here because `oneToMany` is the default.

Indexer output for one-to-many parsing

This Markdown file would result in three search documents after indexing, due to the three content sections. The search document resulting from the first content section of the provided Markdown document would contain the following values for `content`, `sections`, `h1`, and `h2`:

```
HTTP
{
  {
    "content": "Content for section 1.\r\n",
    "sections": {
      "h1": "Section 1",
      "h2": ""
    },
    "ordinal_position": 1
  },
  {
    "content": "Content for subsection 1.1.\r\n",
    "sections": {
      "h1": "Section 1",
      "h2": "Subsection 1.1"
    },
    "ordinal_position": 2
  },
  {
    "content": "Content for section 2.\r\n",
    "sections": {
      "h1": "Section 2",
      "h2": ""
    },
    "ordinal_position": 3
  }
}
```

Map one-to-many fields in a search index

Field mappings associate a source field with a destination field in situations where the field names and types aren't identical. But field mappings can also be used to match parts of a Markdown document and "lift" them into top-level fields of the search document.

The following example illustrates this scenario. For more information about field mappings in general, see [field mappings](#).

Assume a search index with the following fields: `raw_content` of type `Edm.String`, `h1_header` of type `Edm.String`, and `h2_header` of type `Edm.String`. To map your Markdown into the desired shape, use the following field mappings:

```
HTTP
```

```
"fieldMappings" : [
  { "sourceFieldName" : "/content", "targetFieldName" : "raw_content" },
  { "sourceFieldName" : "/sections/h1", "targetFieldName" : "h1_header" },
  { "sourceFieldName" : "/sections/h2", "targetFieldName" : "h2_header" },
]
```

The resulting search document in the index would look as follows:

HTTP

```
{
{
  "raw_content": "Content for section 1.\r\n",
  "h1_header": "Section 1",
  "h2_header": "",
},
{
  "raw_content": "Content for section 1.1.\r\n",
  "h1_header": "Section 1",
  "h2_header": "Subsection 1.1",
},
{
  "raw_content": "Content for section 2.\r\n",
  "h1_header": "Section 2",
  "h2_header": ""
}
```

Use one-to-one parsing mode

In the one-to-one parsing mode, the entire Markdown document is indexed as a single search document, preserving the hierarchy and structure of the original content. This mode is most useful when the files to be indexed share a common structure, so that you can use this common structure in the index to make the relevant fields searchable.

Within the indexer definition, set the `parsingMode` to `"markdown"` and use the optional `markdownHeaderDepth` parameter to define the maximum heading depth for chunking. If not specified, it defaults to `h6`, capturing all possible header depths.

The Markdown is parsed based on headers into search documents, which contain the following content:

- `document_content`: Contains the full Markdown text as a single string. This field serves as a raw representation of the input document.

- `sections`: An array of objects that contains the hierarchical representation of the sections within the Markdown document. Each section is represented as an object within this array and captures the structure of the document in a nested manner corresponding to the headers and their respective content. The fields are accessible through field mappings by referencing the path, for example `/sections/content`. The objects in this array have the following properties:
 - `header_level`: A string that indicates the level of the header (`h1`, `h2`, `h3`, etc.) in Markdown syntax. This field helps in understanding the hierarchy and structuring of the content.
 - `header_name`: A string containing the text of the header as it appears in the Markdown document. This field provides a label or title for the section.
 - `content`: A string containing text content that immediately follows the header, up to the next header. This field captures the detailed information or description associated with the header. If there's no content directly under a header, the value is an empty string.
 - `ordinal_position`: An integer value indicating the position of the section within the document hierarchy. This field is used for ordering the sections in their original sequence as they appear in the document, beginning with an ordinal position of 1 and incrementing sequentially for each content block.
 - `sections`: An array that contains objects representing subsections nested under the current section. This array follows the same structure as the top-level `sections` array, allowing for the representation of multiple levels of nested content. Each subsection object also includes `header_level`, `header_name`, `content`, and `ordinal_position` properties, enabling a recursive structure that represents and hierarchy of the Markdown content.

Here's the sample Markdown that we're using to explain the index schemas designed around each parsing mode.

```
Markdown

# Section 1
Content for section 1.

## Subsection 1.1
Content for subsection 1.1.

# Section 2
Content for section 2.
```

Index schema for one-to-one parsing

If you aren't utilizing field mappings, the shape of the index should reflect the shape of the Markdown content. Given the structure of sample Markdown with its two sections and single subsection, the index should look similar to the following example:

HTTP

```
{  
  "name": "my-markdown-index",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true  
    },  
    {  
      "name": "document_content",  
      "type": "Edm.String"  
    },  
    {  
      "name": "sections",  
      "type": "Collection(Edm.ComplexType)",  
      "fields": [  
        {  
          "name": "header_level",  
          "type": "Edm.String"  
        },  
        {  
          "name": "header_name",  
          "type": "Edm.String"  
        },  
        {  
          "name": "content",  
          "type": "Edm.String"  
        },  
        {  
          "name": "ordinal_position",  
          "type": "Edm.Int32"  
        },  
        {  
          "name": "sections",  
          "type": "Collection(Edm.ComplexType)",  
          "fields": [  
            {  
              "name": "header_level",  
              "type": "Edm.String"  
            },  
            {  
              "name": "header_name",  
              "type": "Edm.String"  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```
        "name": "content",
        "type": "Edm.String"
    },
    {
        "name": "ordinal_position",
        "type": "Edm.Int32"
    }
]
}
}
}
```

Indexer definition for one-to-one parsing

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-08-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name": "my-markdown-indexer",
    "dataSourceName": "my-blob-datasource",
    "targetIndexName": "my-target-index",
    "parameters": {
        "configuration": {
            "parsingMode": "markdown",
            "markdownParsingSubmode": "oneToOne",
        }
    }
}
```

Indexer output for one-to-one parsing

Because the Markdown we want to index only goes to a depth of `h2` ("##"), we need `sections` fields nested to a depth of 2 to match that. This configuration would result in the following data in the index:

HTTP

```
"document_content": "# Section 1\r\nContent for section 1.\r\n## Subsection 1.1\r\nContent for subsection 1.1.\r\n# Section 2\r\nContent for section 2.\r\n",
"sections": [
    {
        "header_level": "h1",
        "header_name": "Section 1",
        "content": "Content for section 1.",
        "ordinal_position": 1,
        "sections": [
            {
                "header_level": "h2",
                "header_name": "Subsection 1.1",
                "content": "Content for subsection 1.1."
            }
        ]
    }
]
```

```

"sections": [
  {
    "header_level": "h2",
    "header_name": "Subsection 1.1",
    "content": "Content for subsection 1.1.",
    "ordinal_position": 2,
  }
],
{
  "header_level": "h1",
  "header_name": "Section 2",
  "content": "Content for section 2.",
  "ordinal_position": 3,
  "sections": []
}
]
}

```

As you can see, the ordinal position increments based on the location of the content within the document.

It should also be noted that if header levels are skipped in the content, then structure of the resulting document reflects the headers that are present in the Markdown content, not necessarily containing nested sections for `h1` through `h6` consecutively. For example, when the document begins at `h2`, then the first element in the top-level sections array is `h2`.

Map one-to-one fields in a search index

If you would like to extract fields with custom names from the document, you can use field mappings to do so. Using the same Markdown sample as before, consider the following index configuration:

HTTP

```

{
  "name": "my-markdown-index",
  "fields": [
    {
      "name": "document_content",
      "type": "Edm.String",
    },
    {
      "name": "document_title",
      "type": "Edm.String",
    },
    {
      "name": "opening_subsection_title"
      "type": "Edm.String",
    }
  ]
}

```

```
        "name": "summary_content",
        "type": "Edm.String",
    }
]
}
```

Extracting specific fields from the parsed Markdown is handled similar to how the document paths are in `outputFieldMappings`, except the path begins with `/sections` instead of `/document`. So, for example, `/sections/0/content` would map to the content under the item at position 0 in the sections array.

An example of a strong use case might look something like this: all Markdown files have a document title in the first `h1`, a subsection title in the first `h2`, and a summary in the content of the final paragraph underneath the final `h1`. You could use the following field mappings to index only that content:

HTTP

```
"fieldMappings" : [
    { "sourceFieldName" : "/content", "targetFieldName" : "raw_content" },
    { "sourceFieldName" : "/sections/0/header_name", "targetFieldName" :
"document_title" },
    { "sourceFieldName" : "/sections/0/sections/header_name", "targetFieldName" :
"opening_subsection_title" },
    { "sourceFieldName" : "/sections/1/content", "targetFieldName" :
"summary_content" },
]
```

Here you would extract only the relevant pieces from that document. To most effectively use this functionality, documents you plan to index should share the same hierarchical header structure.

The resulting search document in the index would look as follows:

HTTP

```
{
    "content": "Content for section 1.\r\n",
    "document_title": "Section 1",
    "opening_subsection_title": "Subsection 1.1",
    "summary_content": "Content for section 2."
}
```

 **Note**

These examples specify how to use these parsing modes entirely with or without field mappings, but you can apply both in one scenario if it suits your needs.

Managing stale documents from Markdown re-indexing

When using one-to-many parsing mode, re-indexing a modified Markdown file can result in stale or duplicate documents if sections are removed. This behavior is specific to one-to-many mode and doesn't apply to one-to-one parsing.

Behavior overview

One-to-many parsing mode

In `oneToMany` mode, each Markdown section (based on headers) is indexed as a separate search document. When the file is re-indexed:

- **No automatic deletion:** The indexer overwrites existing documents with new ones, but it does not delete documents that no longer correspond to any content in the updated file.
- **Potential for duplicates:** This issue specifically arises only when more sections are deleted than inserted between indexing runs. In such cases, leftover documents from the previous version remain in the index, leading to stale entries that no longer reflect the current state of the source file.

One-to-one parsing mode

In `oneToOne` mode, the entire Markdown file is indexed as a single search document. When the file is re-indexed:

- **Overwrite behavior:** The existing document is replaced entirely with the new version.
- **No stale sections:** When the file is re-indexed, the existing document is replaced with the updated version and removed content is no longer included. The only exception is if the file path or blob URI changes, which could result in a new document being created alongside the old one.

Workaround options

To ensure the index reflects the current state of your Markdown files, consider one of the following approaches:

Option 1. Soft delete with metadata

This method uses a soft-delete to delete documents associated with a specific blob. For more information, see [Change and delete detection using indexers for Azure Storage in Azure AI Search](#).

Steps:

1. Mark the blob as deleted by setting a metadata field.
2. Let the indexer run. It deletes all documents in the index associated with that blob.
3. Remove the soft-delete marker and re-index the file.

Option 2. Use the delete API

Before re-indexing a modified Markdown file, explicitly delete the existing documents associated with that file using the [delete API](#). You can either:

- Manually identify individual stale documents by identifying duplicates in the index to be deleted. This may be feasible for small, well-understood changes but can be time-consuming.
- (Recommended) Remove all documents generated from the same parent file before re-indexing, ensuring inconsistencies are avoided.

Steps:

1. Identify the id of the documents associated with the file. Use a query like the following example to retrieve the document key IDs (for example, `id`, `chunk_id`, etc.) for all documents tied to a specific file. Replace `metadata_storage_path` with the appropriate field in your index that maps to the file path or blob URI. This field must be a key.

HTTP

```
GET https://[service name].search.windows.net/indexes/[index name]/docs?api-version=2025-05-01-preview
Content-Type: application/json
api-key: [admin key]
```

```
{
  "filter": "metadata_storage_path eq 'https://<storage-account>.blob.core.windows.net/<container-name>/<file-name>.md'",
  "select": "id"
}
```

2. Issue a delete request for the documents with the identified keys.

HTTP

```
POST https://[service name].search.windows.net/indexes/[index  
name]/docs/index?api-version=2025-05-01-preview  
Content-Type: application/json  
api-key: [admin key]  
  
{  
    "value": [  
        {  
            "@search.action": "delete",  
            "id": "aHR0c...jI1"  
        },  
        {  
            "@search.action": "delete",  
            "id": "aHR0...MQ2"  
        }  
    ]  
}
```

3. Re-index the updated file.

Next steps

- [Configure blob indexers](#)
- [Define field mappings](#)
- [Indexers overview](#)
- [How to index CSV blobs with a blob indexer](#)
- [Tutorial: Search Markdown data from Azure Blob Storage](#)

Indexer troubleshooting guidance for Azure AI Search

Occasionally, indexers run into problems that don't produce errors or that occur on other Azure services, such as during authentication or when connecting. This article focuses on troubleshooting indexer problems when there are no messages to guide you. It also provides troubleshooting for errors that come from non-search resources used during indexing.

(!) Note

If you have an Azure AI Search error to investigate, see [Troubleshooting common indexer errors and warnings](#) instead.

Best practices

These are some best practices and recommendations when working with indexers:

Indexers are designed to run on a schedule

- For reliable indexing, configure your indexers to run on a [regular schedule](#). Scheduled runs automatically pick up any documents missed in previous runs due to transient errors, network interruptions, or temporary service issues. This approach helps maintain data consistency and minimizes the need for manual intervention.
- For [large data sources](#), the initial enumeration and indexing can take hours or even days. Running your indexer on a schedule allows that progress continues and errors are retried automatically. Avoid relying solely on manual or on-demand indexer runs, as these do not provide the same reliability or transient error recovery.

Indexers provide best-effort indexing over time

- Built-in indexers are designed to process all documents without permanent errors over time, if not in the current run, then in subsequent scheduled runs. They offer a convenient, low/no-code way to index data for common scenarios, enabling faster development and easier maintenance. However, if they have AI enrichment capabilities, they are not optimized for very large-scale workloads. For guidance on handling large datasets, see [how to index large data sets](#).
- If your solution requires strict control over indexing timelines, use the Push APIs instead, such as the [Documents Index REST API](#) or the [IndexDocuments method \(Azure SDK for .NET\)](#). These options give you full control of the indexing pipeline.

- Indexers can occasionally fall out of schedule. While this is uncommon and auto-recovery mechanisms exist, recovery may take time. This behavior is expected.

Troubleshoot connections to restricted resources

For data sources under Azure network security, indexers are limited in how they make the connection. Currently, indexers can access restricted data sources [behind an IP firewall](#) or on a virtual network through a [private endpoint](#) using a shared private link.

Error connecting to Azure AI services on a private connection

If you get an error code 403 with the following message, you might have a problem with how the resource endpoint is specified in a skillset:

- "A Virtual Network is configured for this resource. Please use the correct endpoint for making requests. Check <https://aka.ms/cogservc-vnet> for more details."

This error occurs if you have [configured a shared private link](#) for connections to Azure AI services multi-service, and the endpoint is missing a custom subdomain. A custom subdomain is the first part of the endpoint (for example, `http://my-custom-subdomain.cognitiveservices.azure.com`). A custom domain might be missing if you created the resource in Azure AI Foundry.

If the Azure AI services multi-service account isn't in the same region as Azure AI Search, [use a keyless connection](#) when attaching a billable Azure AI resource.

Firewall rules

Azure Storage, Azure Cosmos DB and Azure SQL provide a configurable firewall. There's no specific error message when the firewall blocks the request. Typically, firewall errors are generic. Some common errors include:

- The remote server returned an error: (403) Forbidden
- This request is not authorized to perform this operation
- Credentials provided in the connection string are invalid or have expired

There are two options for allowing indexers to access these resources in such an instance:

- Configure an inbound rule for the IP address of your search service and the IP address range of [AzureCognitiveSearch service tag](#). Details for configuring IP address range restrictions for each data source type can be found from the following links:
 - [Azure Storage](#)

- Azure Cosmos DB
 - Azure SQL
- As a last resort or as a temporary measure, disable the firewall by allowing access from All Networks.

Limitation: IP address range restrictions only work if your search service and your storage account are in different regions.

In addition to data retrieval, indexers also send outbound requests through skillsets and [custom skills](#). For custom skills based on an Azure function, be aware that Azure functions also have [IP address restrictions](#). The list of IP addresses to allow through for custom skill execution include the IP address of your search service and the IP address range of `AzureCognitiveSearch` service tag.

Network security group (NSG) rules

When an indexer accesses data on a SQL managed instance, or when an Azure VM is used as the web service URI for a [custom skill](#), the network security group determines whether requests are allowed in.

For external resources residing on a virtual network, [configure inbound NSG rules](#) for the `AzureCognitiveSearch` service tag.

For more information about connecting to a virtual machine, see [Configure a connection to SQL Server on an Azure VM](#).

Network errors

Usually, network errors are generic. Some common errors include:

- A network-related or instance-specific error occurred while establishing a connection to the server
- The server was not found or was not accessible
- Verify that the instance name is correct and that the source is configured to allow remote connections

When you receive any of those errors:

- Make sure your source is accessible by trying to connect to it directly and not through the search service
- Check your resource in the Azure portal for any current errors or outages
- Check for any network outages in [Azure Status](#) ↗

- Verify you're using a public DNS for name resolution and not an [Azure Private DNS](#)

Azure SQL Database serverless indexing (error code 40613)

If your SQL database is on a [serverless compute tier](#), make sure that the database is running (and not paused) when the indexer connects to it.

If the database is paused, the first sign in from your search service is expected to auto-resume the database, but instead returns an error stating that the database is unavailable, giving error code 40613. After the database is running, retry the sign in to establish connectivity.

Microsoft Entra Conditional Access policies

When you create a SharePoint indexer, there's a step requiring you to sign in to your Microsoft Entra app after providing a device code. If you receive a message that says "Your sign-in was successful but your admin requires the device requesting access to be managed", the indexer is probably blocked from the SharePoint document library by a [Conditional Access](#) policy.

To update the policy and allow indexer access to the document library:

1. Open the Azure portal and search for **Microsoft Entra Conditional Access**.
2. Select **Policies** on the left menu. If you don't have access to view this page, you need to either find someone who has access or get access.
3. Determine which policy is blocking the SharePoint indexer from accessing the document library. The policy that might be blocking the indexer includes the user account that you used to authenticate during the indexer creation step in the **Users and groups** section. The policy also might have **Conditions** that:
 - Restrict **Windows** platforms.
 - Restrict **Mobile apps and desktop clients**.
 - Have **Device state** configured to **Yes**.
4. Once you've confirmed which policy is blocking the indexer, make an exemption for the indexer. Start by retrieving the search service IP address.

First, obtain the fully qualified domain name (FQDN) of your search service. The FQDN looks like `<your-search-service-name>.search.windows.net`. You can find the FQDN in the Azure portal.

The screenshot shows the Azure portal interface for a search service named 'contoso-search-svc'. The left sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Semantic Search (Preview), Knowledge Center, Keys, Scale, Search traffic analytics, Identity, and Properties. The main content area is titled 'Overview' and displays the following information:

- Resource group:** (move)
- Status:** Running
- Location:** West US 2
- Subscription:** (move)
- Pricing tier:** Standard
- Replicas:** 1 (No SLA)
- Partitions:** 1
- Search units:** 1

Below this, there's a 'Tags' section with a link to 'Edit' and a note to 'Click here to add tags'.

At the bottom of the main content area, there are tabs for Get Started, Usage, Monitoring, Indexes, Indexers, Data sources, Skillsets, and more.

Now that you have the FQDN, get the IP address of the search service by performing a `nslookup` (or a `ping`) of the FQDN. In the following example, you would add "150.0.0.1" to an inbound rule on the Azure Storage firewall. It might take up to 15 minutes after the firewall settings have been updated for the search service indexer to be able to access the Azure Storage account.

```
Azure PowerShell

nslookup contoso.search.windows.net
Server: server.example.org
Address: 10.50.10.50

Non-authoritative answer:
Name: <name>
Address: 150.0.0.1
Aliases: contoso.search.windows.net
```

5. Get the IP address ranges for the indexer execution environment for your region.

Extra IP addresses are used for requests that originate from the indexer's [multitenant execution environment](#). You can get this IP address range from the service tag.

The IP address ranges for the `AzureCognitiveSearch` service tag can be either obtained via the [discovery API](#) or the [downloadable JSON file](#).

For this exercise, assuming the search service is the Azure Public cloud, the [Azure Public JSON file](#) should be downloaded.

Azure IP Ranges and Service Tags – Public Cloud

Important! Selecting a language below will dynamically change the complete page content to that language.

Language:

English

[Download](#)

Azure IP Ranges and Service Tags – Public Cloud

 [Details](#)

 [System Requirements](#)

 [Install Instructions](#)

From the JSON file, assuming the search service is in West Central US, the list of IP addresses for the multitenant indexer execution environment are listed below.

JSON

```
{  
  "name": "AzureCognitiveSearch.WestCentralUS",  
  "id": "AzureCognitiveSearch.WestCentralUS",  
  "properties": {  
    "changeNumber": 1,  
    "region": "westcentralus",  
    "platform": "Azure",  
    "systemService": "AzureCognitiveSearch",  
    "addressPrefixes": [  
      "52.150.139.0/26",  
      "52.253.133.74/32"  
    ]  
  }  
}
```

6. Back on the Conditional Access page in Azure portal, select **Named locations** from the menu on the left, then select **+ IP ranges location**. Give your new named location a name and add the IP ranges for your search service and indexer execution environments that you collected in the last two steps. 1

- For your search service IP address, you might need to add "/32" to the end of the IP address since it only accepts valid IP ranges.
- Remember that for the indexer execution environment IP ranges, you only need to add the IP ranges for the region that your search service is in.

7. Exclude the new Named location from the policy:
 - a. Select **Policies** on the left menu.
 - b. Select the policy that is blocking the indexer.
 - c. Select **Conditions**.
 - d. Select **Locations**.
 - e. Select **Exclude** then add the new Named location.
 - f. **Save** the changes.
8. Wait a few minutes for the policy to update and enforce the new policy rules.
9. Attempt to create the indexer again:
 - a. Send an update request for the data source object that you created.
 - b. Resend the indexer create request. Use the new code to sign in, then send another indexer creation request.

Indexing unsupported document types

If you're indexing content from Azure Blob Storage, and the container includes blobs of an [unsupported content type](#), the indexer skips that document. In other cases, there might be problems with individual documents.

In this situation, you can [set configuration options](#) to allow indexer processing to continue if there are problems with individual documents.

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "failOnUnsupportedContentType" : false,
    "failOnUnprocessableDocument" : false } }
}
```

Missing documents

Indexers extract documents or rows from an external [data source](#) and create *search documents*, which are then indexed by the search service. Occasionally, a document that exists in data source fails to appear in a search index. This unexpected result can occur due to the following reasons:

- The document was updated after the indexer was run. If your indexer is on a [schedule](#), it eventually reruns and picks up the document.
- The indexer timed out before the document could be ingested. There are [maximum processing time limits](#) after which no documents are processed. You can check indexer status in the Azure portal or by calling [Get Indexer Status \(REST API\)](#).
- [Field mappings](#) or [AI enrichment](#) have changed the document and its articulation in the search index is different from what you expect.
- Change tracking values are erroneous or prerequisites are missing. If your high watermark value is a date set to a future time, then any documents that have an earlier date are skipped by the indexer. You can determine your indexer's change tracking state using the 'initialTrackingState' and 'finalTrackingState' fields in the [indexer status](#). Indexers for Azure SQL and MySQL must have an index on the high water mark column of the source table, or queries used by the indexer might time out.

Tip

If documents are missing, check the [query](#) you're using to make sure it isn't excluding the document in question. To query for a specific document, use the [Lookup Document REST API](#).

Missing content from Blob Storage

The blob indexer [finds and extracts text from blobs in a container](#). Some problems with extracting text include:

- The document only contains scanned images. PDF blobs that have non-text content, such as scanned images (JPGs), don't produce results in a standard blob indexing pipeline. If you have image content with text elements, you can use [OCR or image analysis](#) to find and extract the text.
- The blob indexer is configured to only index metadata. To extract content, the blob indexer must be configured to [extract both content and metadata](#):

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
  ... other parts of indexer definition
```

```
"parameters" : { "configuration" : { "dataToExtract" : "contentAndMetadata" } }
```

Missing content from Azure Cosmos DB

Azure AI Search has an implicit dependency on Azure Cosmos DB indexing. If you turn off automatic indexing in Azure Cosmos DB, Azure AI Search returns a successful state, but fails to index container contents. For instructions on how to check settings and turn on indexing, see [Manage indexing in Azure Cosmos DB](#).

Document count discrepancy between the data source and index

An indexer might show a different document count than either the data source, the index itself, or count in your code. Here are some possible reasons why this behavior can occur:

- The index can lag in showing the real document count, especially in the Azure portal.
- The indexer has a Deleted Document Policy. The deleted documents get counted by the indexer if the documents are indexed before they get deleted.
- If the ID column in the data source isn't unique. This applies to data sources that have the concept of columns, such as Azure Cosmos DB.
- If the data source definition has a different query than the one you're using to estimate the number of records. In example, in your database, you're querying the database record count, while in the data source definition query, you might be selecting just a subset of records to index.
- The counts are being checked at different intervals for each component of the pipeline: data source, indexer and index.
- The data source has a file that's mapped to many documents. This condition can occur when [indexing blobs](#) and "parsingMode" is set to `jsonArray` and `jsonLines`.

Documents processed multiple times

Indexers use a conservative buffering strategy to ensure that every new and changed document in the data source is picked up during indexing. In certain situations, these buffers can overlap, causing an indexer to index a document two or more times resulting in the processed documents count to be more than actual number of documents in the data source. This behavior does **not** affect the data stored in the index, such as duplicating documents, only that it can take longer to reach eventual consistency. This condition is especially prevalent if any of the following criteria are true:

- On-demand indexer requests are issued in quick succession
- The data source's topology includes multiple replicas and partitions (one such example is discussed [here](#))
- The data source is an Azure SQL database and the column chosen as "high water mark" is of type `datetime2`

Indexers aren't intended to be invoked multiple times in quick succession. If you need updates quickly, the supported approach is to push updates to the index while simultaneously updating the data source. For on-demand processing, we recommend that you pace your requests in five-minute intervals or more, and run the indexer on a schedule.

Example of duplicate document processing with 30 second buffer

Conditions under which a document is processed twice is explained in the following timeline that notes each action and counter action. The following timeline illustrates the issue:

 Expand table

Timeline (hh:mm:ss)	Event	Indexer	Comment
	High Water Mark		
00:01:00	Write <code>doc1</code> to data source with eventual consistency	null	Document timestamp is 00:01:00.
00:01:05	Write <code>doc2</code> to data source with eventual consistency	null	Document timestamp is 00:01:05.
00:01:10	Indexer starts	null	
00:01:11	Indexer queries for all changes before 00:01:10; the replica that the indexer queries happens to be only aware of <code>doc2</code> ; only <code>doc2</code> is retrieved	null	Indexer requests all changes before starting timestamp but actually receives a subset. This behavior necessitates the look back buffer period.
00:01:12	Indexer processes <code>doc2</code> for the first time	null	
00:01:13	Indexer ends	00:01:10	High water mark is updated to starting timestamp of current indexer execution.
00:01:20	Indexer starts	00:01:10	

Timeline (hh:mm:ss)	Event	Indexer High Water Mark	Comment
00:01:21	Indexer queries for all changes between 00:00:40 and 00:01:20; the replica that the indexer queries happens to be aware of both <code>doc1</code> and <code>doc2</code> ; retrieves <code>doc1</code> and <code>doc2</code>	00:01:10	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:22	Indexer processes <code>doc1</code> for the first time	00:01:10	
00:01:23	Indexer processes <code>doc2</code> for the second time	00:01:10	
00:01:24	Indexer ends	00:01:20	High water mark is updated to starting timestamp of current indexer execution.
00:01:32	Indexer starts	00:01:20	
00:01:33	Indexer queries for all changes between 00:00:50 and 00:01:32; retrieves <code>doc1</code> and <code>doc2</code>	00:01:20	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:34	Indexer processes <code>doc1</code> for the second time	00:01:20	
00:01:35	Indexer processes <code>doc2</code> for the third time	00:01:20	
00:01:36	Indexer ends	00:01:32	High water mark is updated to starting timestamp of current indexer execution.
00:01:40	Indexer starts	00:01:32	
00:01:41	Indexer queries for all changes between 00:01:02 and 00:01:40; retrieves <code>doc2</code>	00:01:32	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:42	Indexer processes <code>doc2</code> for the fourth time	00:01:32	
00:01:43	Indexer ends	00:01:40	Notice this indexer execution started more than 30 seconds after the last write to the data source and also processed <code>doc2</code> . This is

Timeline (hh:mm:ss)	Event	Indexer	Comment
		High Water Mark	the expected behavior because if all indexer executions before 00:01:35 are eliminated, this becomes the first and only execution to process <code>doc1</code> and <code>doc2</code> .

In practice, this scenario only happens when on-demand indexers are manually invoked within minutes of each other, for certain data sources. It can result in mismatched numbers (like the indexer processed 345 documents total according to the indexer execution stats, but there are 340 documents in the data source and index) or potentially increased billing if you're running the same skills for the same document multiple times. Running an indexer using a schedule is the preferred recommendation.

Parallel indexing

When multiple indexers are operating simultaneously, it's typical for some to enter a queue, waiting for available resources to begin execution. The number of indexers that can run concurrently depends on several factors. If the indexers aren't linked with [skillsets](#), the capacity to run in parallel relies on the number of [replicas and partitions](#) set up in the AI Search service.

On the other hand, if an indexer is associated with a skillset, it operates within the AI Search's internal clusters. The ability to run concurrently in this case is determined by the complexity of the skillset and whether other skillsets are running simultaneously. Built-in indexers are designed to reliably extract data from the source, so no data is missed if running on a schedule. However, it's expected that the indexer processes of parallelization and scaling out require some time to complete.

Indexing documents with sensitivity labels

If you have [sensitivity labels set on documents](#), you might not be able to index them. If you're getting errors, remove the labels prior to indexing.

See also

- [Troubleshooting common indexer errors and warnings](#)
- [Monitor indexer-based indexing](#)
- [Index large data sets](#)

Last updated on 10/23/2025

Troubleshooting common indexer errors and warnings in Azure AI Search

10/14/2025

This article provides information and solutions to common errors and warnings you might encounter during indexing and AI enrichment in Azure AI Search.

Indexing stops when the error count exceeds '[maxFailedItems](#)'.

If you want indexers to ignore these errors (and skip over "failed documents"), consider updating the `maxFailedItems` and `maxFailedItemsPerBatch` as described [here](#).

➊ Note

Each failed document along with its document key (when available) will show up as an error in the indexer execution status. You can utilize the [index api](#) to manually upload the documents at a later point if you have set the indexer to tolerate failures.

The error information in this article can help you resolve errors, allowing indexing to continue.

Warnings don't stop indexing, but they do indicate conditions that could result in unexpected outcomes. Whether you take action or not depends on the data and your scenario.

Where can you find specific indexer errors?

To verify an indexer status and identify errors in the Azure portal, follow the steps below:

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. On the left, expand **Search Management > Indexers** and select an indexer.
3. Under **Execution History**, select the status. All statuses, including Success, have details about the execution.
4. If there's an error, hover over the error message. A pane appears on the right side of your screen displaying detailed information about the error.

Transient errors

For various reasons, such as transient network communication interruptions, timeouts from long-running processes, or specific document nuances, it's common to encounter transient

errors or warnings during indexer runs. However, these errors are temporary and should be resolved in subsequent indexer runs.

To manage these errors effectively, we recommend [putting your indexer on a schedule](#), for instance, to run every five minutes, where the next run commences five minutes after completing the first run, adhering to the [maximum runtime limit](#) on your service. Regularly scheduled runs help rectify transient errors or warnings.

If an error persists over multiple indexer runs, it's likely not a transient issue. In such cases, refer to the list below for potential solutions.

Error properties

 [Expand table](#)

Property	Description	Example
Key	The ID of the document impacted by the error or warning.	Azure Storage example, where the default ID is the metadata storage path: <code>https://<storageaccount>.blob.core.windows.net/jfk-1k/docid-32112954.pdf</code>
Name	The operation causing the error or warning. This is generated by the following structure: <code>[category]. [subcategory]. [resourceType]. [resourceName]</code>	<code>DocumentExtraction.azureblob.myBlobContainerName</code> <code>Enrichment.WebApiSkill.mySkillName</code> <code>Projection.SearchIndex.OutputFieldMapping.myOutputFieldName</code> <code>Projection.SearchIndex.MergeOrUpload.myIndexName</code> <code>Projection.KnowledgeStore.Table.myTableName</code>
Message	A high-level description of the error or warning.	<code>Could not execute skill because the Web Api request failed.</code>

Property	Description	Example
Details	Specific information that might be helpful in diagnosing the issue, such as the WebApi response if executing a custom skill failed.	<code>link-cryptonyms-list - Error processing the request record : System.ArgumentNullException: Value cannot be null. Parameter name: source at System.Linq.Enumerable.All[TSource](IEnumerable`1 source, Func`2 predicate) at Microsoft.CognitiveSearch.WebApiSkills.JfkWebApiSkills. ...rest of stack trace...</code>
DocumentationLink	A link to relevant documentation with detailed information to debug and resolve the issue. This link will often point to one of the below sections on this page.	https://go.microsoft.com/fwlink/?linkid=2106475

Error: Could not read document

Indexer was unable to read the document from the data source. This can happen due to:

[Expand table](#)

Reason	Details/Example	Resolution
Inconsistent field types across different documents	<code>Type of value has a mismatch with column type. Couldn't store '{47.6,-122.1}' in authors column. Expected type is JArray. Error converting data type nvarchar to float. Conversion failed when converting the nvarchar value '12 months' to data type int. Arithmetic overflow error converting expression to data type int.</code>	Ensure that the type of each field is the same across different documents. For example, if the first document 'startTime' field is a DateTime, and in the second document it's a string, this error is hit.
Errors from the data source's underlying service	From Azure Cosmos DB: <code>{"Errors": ["Request rate is large"]}</code>	Check your storage instance to ensure it's healthy. You might need to adjust your scaling or partitioning.

Reason	Details/Example	Resolution
Transient issues	A transport-level error has occurred when receiving results from the server. (provider: TCP Provider, error: 0 - An existing connection was forcibly closed by the remote host)	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.

Error: Could not extract content or metadata from your document

Indexer with a Blob data source was unable to extract the content or metadata from the document (for example, a PDF file). This can happen due to:

[+] [Expand table](#)

Reason	Details/Example	Resolution
Blob is over the size limit	Document is '150441598' bytes, which exceeds the maximum size '134217728' bytes for document extraction for your current service tier.	Blob indexing errors
Blob has unsupported content type	Document has unsupported content type 'image/png'	Blob indexing errors
Blob is encrypted	Document could not be processed - it may be encrypted or password protected.	You can skip the blob with blob settings .
Transient issues	Error processing blob: The request was aborted: The request was canceled. Document timed out during processing.	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.

Error: Could not parse document

Indexer read the document from the data source, but there was an issue converting the document content into the specified field mapping schema. This can happen due to:

[+] [Expand table](#)

Reason	Details/Example	Resolution
The document key is missing	Document key cannot be missing or empty	Ensure all documents have valid document keys. The document key is determined by setting the 'key' property as part of the index definition . Indexers emit this error when the property flagged as the 'key' can't be found on a particular document.
The document key is invalid	Invalid document key. Keys can only contain letters, digits, underscore (_), dash (-), or equal sign (=).	Ensure all documents have valid document keys. Review Indexing Blob Storage for more details. If you're using the blob indexer, and your document key is the <code>metadata_storage_path</code> field, make sure that the indexer definition has a base64Encode mapping function with <code>parameters</code> equal to <code>null</code> , instead of the path in plain text.
The document key is invalid	Document key cannot be longer than 1024 characters	Modify the document key to meet the validation requirements.
Couldn't apply field mapping to a field	Could not apply mapping function 'functionName' to field 'fieldName'. Array cannot be null. Parameter name: bytes	Double check the field mappings defined on the indexer, and compare with the data of the specified field of the failed document. It might be necessary to modify the field mappings or the document data.
Couldn't read field value	Could not read the value of column 'fieldName' at index 'fieldIndex'. A transport-level error has occurred when receiving results from the server. (provider: TCP Provider, error: 0 - An existing connection was forcibly closed by the remote host.)	These errors are typically due to unexpected connectivity issues with the data source's underlying service. Try running the document through your indexer again later.

Error: Could not map output field 'xyz' to search index due to deserialization problem while applying mapping function 'abc'

The output mapping might have failed because the output data is in the wrong format for the mapping function you're using. For example, applying `Base64Encode` mapping function on binary data would generate this error. To resolve the issue, either rerun indexer without

specifying mapping function or ensure that the mapping function is compatible with the output field data type. See [Output field mapping](#) for details.

Error: Could not execute skill

The indexer wasn't able to run a skill in the skillset.

 [Expand table](#)

Reason	Details/Example	Resolution
Transient connectivity issues	A transient error occurred. Try again later.	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.
Potential product bug	An unexpected error occurred.	This indicates an unknown class of failure and can indicate a product bug. File a support ticket to get help.
A skill has encountered an error during execution	(From Merge Skill) One or more offset values were invalid and couldn't be parsed. Items were inserted at the end of the text	Use the information in the error message to fix the issue. This kind of failure requires action to resolve.

Error: Could not execute skill because the Web API request failed

The skill execution failed because the call to the Web API failed. Typically, this class of failure occurs when custom skills are used, in which case you need to debug your custom code to resolve the issue. If instead the failure is from a built-in skill, refer to the error message for help with fixing the issue.

While debugging this issue, be sure to pay attention to any [skill input warnings](#) for this skill. Your Web API endpoint might be failing because the indexer is passing it unexpected input.

Error: Could not execute skill because Web API skill response is invalid

The skill execution failed because the call to the Web API returned an invalid response. Typically, this class of failure occurs when custom skills are used, in which case you need to

debug your custom code to resolve the issue. If instead the failure is from a built-in skill, file a [support ticket](#) to get assistance.

Error: Type of value has a mismatch with column type. Couldn't store in 'xyz' column. Expected type is 'abc'

If your data source has a field with a different data type than the field you're trying to map in your index, you might encounter this error. Check your data source field data types and make sure they're [mapped correctly to your index data types](#).

Error: Skill did not execute within the time limit

There are two cases under which you might encounter this error message, each of which should be treated differently. Follow the instructions below depending on what skill returned this error for you.

Built-in Azure AI services skills

Many of the built-in cognitive skills, such as language detection, entity recognition, or OCR, are backed by an Azure AI services API endpoint. Sometimes there are transient issues with these endpoints and a request will time out. For transient issues, there's no remedy except to wait and try again. As a mitigation, consider setting your indexer to [run on a schedule](#). Scheduled indexing picks up where it left off. Assuming transient issues are resolved, indexing and cognitive skill processing should be able to continue on the next scheduled run.

If you continue to see this error on the same document for a built-in cognitive skill, file a [support ticket](#) to get assistance, as this isn't expected.

Custom skills

If you encounter a timeout error with a custom skill, there are a couple of things you can try. First, review your custom skill and ensure that it's not getting stuck in an infinite loop and that it's returning a result consistently. Once you have confirmed that a result is returned, check the duration of execution. If you didn't explicitly set a `timeout` value on your custom skill definition, then the default `timeout` is 30 seconds. If 30 seconds isn't long enough for your skill to

execute, you can specify a higher `timeout` value on your custom skill definition. Here's an example of a custom skill definition where the timeout is set to 90 seconds:

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
    "uri": "<your custom skill uri>",  
    "batchSize": 1,  
    "timeout": "PT90S",  
    "context": "/document",  
    "inputs": [  
        {  
            "name": "input",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "output",  
            "targetName": "output"  
        }  
    ]  
}
```

The maximum value that you can set for the `timeout` parameter is 230 seconds. If your custom skill is unable to execute consistently within 230 seconds, you might consider reducing the `batchSize` of your custom skill so that it has fewer documents to process within a single execution. If you have already set your `batchSize` to 1, you need to rewrite the skill to be able to execute in under 230 seconds, or otherwise split it into multiple custom skills so that the execution time for any single custom skill is a maximum of 230 seconds. Review the [custom skill documentation](#) for more information.

Error: Could not 'MergeOrUpload' | 'Delete' document to the search index

The document was read and processed, but the indexer couldn't add it to the search index. This can happen due to:

[+] Expand table

Reason	Details/Example	Resolution
A field contains a term that is too large	A term in your document is larger than the 32-KB limit	You can avoid this restriction by ensuring the field isn't configured as

Reason	Details/Example	Resolution
		filterable, facetable, or sortable.
Document is too large to be indexed	A document is larger than the maximum API request size	How to index large data sets
Document contains too many objects in collection	A collection in your document exceeds the maximum elements across all complex collections limit . The document with key '1000052' has '4303' objects in collections (JSON arrays). At most '3000' objects are allowed to be in collections across the entire document. Remove objects from collections and try indexing the document again.	We recommend reducing the size of the complex collection in the document to below the limit and avoid high storage utilization.
Trouble connecting to the target index (that persists after retries) because the service is under other load, such as querying or indexing.	Failed to establish connection to update index. Search service is under heavy load.	Scale up your search service
Search service is being patched for service update, or is in the middle of a topology reconfiguration.	Failed to establish connection to update index. Search service is currently down/Search service is undergoing a transition.	Configure service with at least three replicas for 99.9% availability per SLA documentation
Failure in the underlying compute/networking resource (rare)	Failed to establish connection to update index. An unknown failure occurred.	Configure indexers to run on a schedule to pick up from a failed state.
An indexing request made to the target index wasn't acknowledged within a timeout period due to network issues.	Couldn't establish connection to the search index in a timely manner.	Configure indexers to run on a schedule to pick up from a failed state. Additionally, try lowering the indexer batch size if this error condition persists.

Error: Could not index document because some of the document's data was not valid

The document was read and processed by the indexer, but due to a mismatch in the configuration of the index fields and the data extracted and processed by the indexer, it couldn't be added to the search index. This can happen due to:

Reason	Details/Example
Data type of one or more fields extracted by the indexer is incompatible with the data model of the corresponding target index field.	The data field '_data_' in the document with key '888' has an invalid value 'of type 'Edm.String''. The expected type was 'Collection(Edm.String)'.
Failed to extract any JSON entity from a string value.	Could not parse value 'of type 'Edm.String'' of field '_data_' as a JSON object. Error:'After parsing a value an unexpected character was encountered: '''. Path '_path_', line 1, position 3162.'
Failed to extract a collection of JSON entities from a string value.	Could not parse value 'of type 'Edm.String'' of field '_data_' as a JSON array. Error:'After parsing a value an unexpected character was encountered: '''. Path '[0]', line 1, position 27.'
An unknown type was discovered in the source document.	Unknown type '_unknown_' cannot be indexed
An incompatible notation for geography points was used in the source document.	WKT POINT string literals are not supported. Use GeoJson point literals instead

In all these cases, refer to [Supported Data types](#) and [Data type map for indexers](#) to make sure that you build the index schema correctly and have set up appropriate [indexer field mappings](#). The error message includes details that can help track down the source of the mismatch.

Error: Integrated change tracking policy cannot be used because table has a composite primary key

This applies to SQL tables, and usually happens when the key is either defined as a composite key or, when the table has defined a unique clustered index (as in a SQL index, not an Azure Search index). The main reason is that the key attribute is modified to be a composite primary key in a [unique clustered index](#). In that case, make sure that your SQL table doesn't have a unique clustered index, or that you map the key field to a field that is guaranteed not to have duplicate values.

Error: Could not process document within indexer max run time

This error occurs when the indexer is unable to finish processing a single document from the data source within the allowed execution time. [Maximum running time](#) is shorter when skillsets are used. When this error occurs, if you have maxFailedItems set to a value other than 0, the indexer bypasses the document on future runs so that indexing can progress. If you can't afford to skip any document, or if you're seeing this error consistently, consider breaking documents into smaller documents so that partial progress can be made within a single indexer execution.

Error: Could not project document

This error occurs when the indexer is attempting to [project data into a knowledge store](#) and there was a failure on the attempt. This failure could be consistent and fixable, or it could be a transient failure with the projection output sink that you might need to wait and retry in order to resolve. Here's a set of known failure states and possible resolutions.

[] [Expand table](#)

Reason	Details/Example	Resolution
Couldn't update projection blob <code>'blobUri'</code> in container <code>'containerName'</code>	The specified container doesn't exist.	The indexer checks if the specified container has been previously created and will create it if necessary, but it only performs this check once per indexer run. This error means that something deleted the container after this step. To resolve this error, try this: leave your storage account information alone, wait for the indexer to finish, and then rerun the indexer.
Couldn't update projection blob <code>'blobUri'</code> in container <code>'containerName'</code>	Unable to write data to the transport connection: An existing connection was forcibly closed by the remote host.	This is expected to be a transient failure with Azure Storage and thus should be resolved by rerunning the indexer. If you encounter this error consistently, file a support ticket so it can be investigated further.
Couldn't update row <code>'projectionRow'</code> in table <code>'tableName'</code>	The server is busy.	This is expected to be a transient failure with Azure Storage and thus should be resolved by rerunning the indexer. If you encounter this error consistently, file a support ticket so it can be investigated further.

Error: The cognitive service for skill '<skill-name>' has been throttled

Skill execution failed because the call to Azure AI services was throttled. Typically, this class of failure occurs when too many skills are executing in parallel. If you're using the Microsoft.Search.Documents client library to run the indexer, you can use the [SearchIndexingBufferedSender](#) to get automatic retry on failed steps. Otherwise, you can reset and rerun the indexer.

Error: Expected IndexAction metadata

An 'Expected IndexAction metadata' error means when the indexer attempted to read the document to identify what action should be taken, it didn't find any corresponding metadata on the document. Typically, this error occurs when the indexer has an annotation cache added or removed without resetting the indexer. To address this, you should [reset and rerun the indexer](#).

Warning: Skill input was invalid

An input to the skill was missing, it has the wrong type, or otherwise, invalid. You might see the following information:

- Could not execute skill
- Skill executed but may have unexpected results

Cognitive skills have required inputs and optional inputs. For example, the [Key phrase extraction skill](#) has two required inputs `text`, `languageCode`, and no optional inputs. Custom skill inputs are all considered optional inputs.

If necessary inputs are missing or if the input isn't the right type, the skill gets skipped and generates a warning. Skipped skills don't generate outputs. If downstream skills consume the outputs of the skipped skill, they can generate other warnings.

If an optional input is missing, the skill still runs, but it might produce unexpected output due to the missing input.

In both cases, this warning is due to the shape of your data. For example, if you have a document containing information about people with the fields `firstName`, `middleName`, and `lastName`, you might have some documents that don't have an entry for `middleName`. If you pass `middleName` as an input to a skill in the pipeline, then it's expected that this skill input is missing some of the time. You need to evaluate your data and scenario to determine whether or not any action is required as a result of this warning.

If you want to provide a default value for a missing input, you can use the [Conditional skill](#) to generate a default value and then use the output of the [Conditional skill](#) as the skill input.

JSON

```
{ "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill", "context": "/document", "inputs": [ { "name": "condition", "source": "= $(/document/language) == null" }, { "name": "whenTrue", "source": "= 'en'" }, { "name": "whenFalse", "source": "= $(/document/language)" } ], "outputs": [ { "name": "output", "targetName": "languageWithDefault" } ] }
```

[] [Expand table](#)

Reason	Details/Example	Resolution
Skill input is the wrong type	"Required skill input was not of the expected type <code>String</code> . Name: <code>text</code> , Source: <code>/document/merged_content</code> ." "Required skill input wasn't of the expected format. Name: <code>text</code> , Source: <code>/document/merged_content</code> ." "Cannot iterate over non-array <code>/document/normalized_images/0/imageCelebrities/0/detail/celebrities</code> ." "Unable to select <code>0</code> in non-array <code>/document/normalized_images/0/imageCelebrities/0/detail/celebrities</code> "	Certain skills expect inputs of particular types, for example Sentiment skill expects <code>text</code> to be a string. If the input specifies a nonstring value, then the skill doesn't execute and generates no outputs. Ensure your data set has input values uniform in type, or use a Custom Web API skill to preprocess the input. If you're iterating the skill over an array, check the skill

Reason	Details/Example	Resolution
		context and input have * in the correct positions. Usually both the context and input source should end with * for arrays.
Skill input is missing	Required skill input is missing. Name: text, Source: /document/merged_content Missing value /document/normalized_images/0/imageTags. Unable to select 0 in array /document/pages of length 0.	If this warning occurs for all documents, there could be a typo in the input paths. Check the property name casing. Check for an extra or missing * in the path. Verify that the documents from the data source provide the required inputs.
Skill language code input is invalid	Skill input languageCode has the following language codes X,Y,Z, at least one of which is invalid.	See more details below.

Warning: Skill input 'languageCode' has the following language codes 'X,Y,Z', at least one of which is invalid.

One or more of the values passed into the optional languageCode input of a downstream skill isn't supported. This can occur if you're passing the output of the [LanguageDetectionSkill](#) to subsequent skills, and the output consists of more languages than are supported in those downstream skills.

Note that you can also get a warning similar to this one if an invalid `countryHint` input gets passed to the `LanguageDetectionSkill`. If that happens, validate that the field you're using from your data source for that input contains valid ISO 3166-1 alpha-2 two letter country codes. If some are valid and some are invalid, continue with the following guidance but replace `languageCode` with `countryHint` and `defaultLanguageCode` with `defaultCountryHint` to match your use case.

If you know that your data set is all in one language, you should remove the `LanguageDetectionSkill` and the `languageCode` skill input and use the `defaultLanguageCode` skill parameter for that skill instead, assuming the language is supported for that skill.

If you know that your data set contains multiple languages and thus you need the `LanguageDetectionSkill` and `languageCode` input, consider adding a `ConditionalSkill` to filter out the text with languages that aren't supported before passing in the text to the downstream skill. Here's an example of what this might look like for the `EntityRecognitionSkill`:

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",  
    "context": "/document",  
    "inputs": [  
        { "name": "condition", "source": "= $(/document/language) == 'de' ||  
        $(/document/language) == 'en' || $(/document/language) == 'es' ||  
        $(/document/language) == 'fr' || $(/document/language) == 'it'" },  
        { "name": "whenTrue", "source": "/document/content" },  
        { "name": "whenFalse", "source": "= null" }  
    "outputs": [ { "name": "output", "targetName":  
    "supportedByEntityRecognitionSkill" } ]  
}
```

Here are some references for the currently supported languages for each of the skills that can produce this error message:

- [EntityRecognitionSkill supported languages](#)
- [EntityLinkingSkill supported languages](#)
- [KeyPhraseExtractionSkill supported languages](#)
- [LanguageDetectionSkill supported languages](#)
- [PIIDetectionSkill supported languages](#)
- [SentimentSkill supported languages](#)
- [Translator supported languages](#)
- [Text SplitSkill supported languages](#): da, de, en, es, fi, fr, it, ko, pt

Warning: Skill input was truncated

Cognitive skills limit the length of text that can be analyzed at one time. If the text input exceeds the limit, the text is truncated before it's enriched. The skill executes, but not over all of your data.

In the example `LanguageDetectionSkill` below, the `'text'` input field might trigger this warning if the input is over the character limit. Input limits can be found in the [skills reference documentation](#).

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/text"  
        }  
    ],  
    "outputs": [...]  
}
```

If you want to ensure that all text is analyzed, consider using the [Split skill](#).

Warning: Web API skill response contains warnings

The indexer ran the skill in the skillset, but the response from the Web API request indicates there are warnings. Review the warnings to understand how your data is impacted and whether further action is required.

Warning: The current indexer configuration does not support incremental progress

This warning only occurs for Azure Cosmos DB data sources.

Incremental progress during indexing ensures that if indexer execution is interrupted by transient failures or execution time limit, the indexer can pick up where it left off next time it runs, instead of having to reindex the entire collection from scratch. This is especially important when indexing large collections.

The ability to resume an unfinished indexing job is predicated on having documents ordered by the `_ts` column. The indexer uses the timestamp to determine which document to pick up

next. If the `_ts` column is missing or if the indexer can't determine if a custom query is ordered by it, the indexer starts at beginning and you'll see this warning.

It's possible to override this behavior, enabling incremental progress and suppressing this warning by using the `assumeOrderByHighWaterMarkColumn` configuration property.

For more information, see [Incremental progress and custom queries](#).

Warning: Some data was lost during projection.

Row 'X' in table 'Y' has string property 'Z' which was too long.

The [Table Storage service](#) has limits on how large [entity properties](#) can be. Strings can have 32,000 characters or less. If a row with a string property longer than 32,000 characters is being projected, only the first 32,000 characters are preserved. To work around this issue, avoid projecting rows with string properties longer than 32,000 characters.

Warning: Truncated extracted text to X characters

Indexers limit how much text can be extracted from any one document. This limit depends on the pricing tier: 32,000 characters for Free tier, 64,000 for Basic, 4 million for Standard, 8 million for Standard S2, and 16 million for Standard S3. Text that was truncated won't be indexed. To avoid this warning, try breaking apart documents with large amounts of text into multiple, smaller documents.

For more information, see [Indexer limits](#).

Warning: Could not map output field 'X' to search index

Output field mappings that reference non-existent/null data will produce warnings for each document and result in an empty index field. To work around this issue, double-check your output field-mapping source paths for possible typos, or set a default value using the [Conditional skill](#). See [Output field mapping](#) for details.

Reason	Details/Example	Resolution
Can't iterate over non-array	"Cannot iterate over non-array /document/normalized_images/0/imageCelebrities/0/detail/celebrities."	This error occurs when the output isn't an array. If you think the output should be an array, check the indicated output source field path for errors. For example, you might have a missing or extra <code>*</code> in the source field name. It's also possible that the input to this skill is null, resulting in an empty array. Find similar details in Skill Input was Invalid section.
Unable to select <code>0</code> in non-array	"Unable to select <code>0</code> in non-array /document/pages."	This could happen if the skills output doesn't produce an array and the output source field name has array index or <code>*</code> in its path. Double check the paths provided in the output source field names and the field value for the indicated field name. Find similar details in Skill Input was Invalid section.

Warning: The data change detection policy is configured to use key column 'X'

Data change detection policies have specific requirements for the columns they use to detect change. One of these requirements is that this column is updated every time the source item is changed. Another requirement is that the new value for this column is greater than the previous value. Key columns don't fulfill this requirement because they don't change on every update. To work around this issue, select a different column for the change detection policy.

Warning: Document text appears to be UTF-16 encoded, but is missing a byte order mark

The [indexer parsing modes](#) need to know how text is encoded before parsing it. The two most common ways of encoding text are UTF-16 and UTF-8. UTF-8 is a variable-length encoding where each character is between 1 byte and 4 bytes long. UTF-16 is a fixed-length encoding where each character is 2 bytes long. UTF-16 has two different variants, `big endian` and `little endian`. Text encoding is determined by a `byte order mark`, a series of bytes before the text.

[] [Expand table](#)

Encoding	Byte Order Mark
UTF-16 Big Endian	0xFE 0xFF
UTF-16 Little Endian	0xFF 0xFE
UTF-8	0xEF 0xBB 0xBF

If no byte order mark is present, the text is assumed to be encoded as UTF-8.

To work around this warning, determine what the text encoding for this blob is and add the appropriate byte order mark.

Warning: Azure Cosmos DB collection 'X' has a Lazy indexing policy. Some data may be lost

Collections with [Lazy](#) indexing policies can't be queried consistently, resulting in your indexer missing data. To work around this warning, change your indexing policy to Consistent.

Warning: The document contains very long words (longer than 64 characters). These words may

result in truncated and/or unreliable model predictions.

This warning is passed from the Language service of Azure AI services. In some cases, it's safe to ignore this warning, for example if the long string is just a long URL. Be aware that when a word is longer than 64 characters, it's truncated to 64 characters which can affect model predictions.

Error: Cannot write more bytes to the buffer than the configured maximum buffer size

Indexers have [document size limits](#). Make sure that the documents in your data source are smaller than the supported size limit, as documented for your service tier.

Error: Failed to compare value 'X' of type M to value 'Y' of type N.

This error usually happens in Azure SQL indexers when the source column type used for [dataChangeDetectionPolicy](#) doesn't match what the indexer expects, especially if [convertHighWaterMarkToRowVersion](#) is turned on.

For example, if the column used for change detection is of type datetime, but the indexer expects a rowversion type because [convertHighWaterMarkToRowVersion](#) is enabled, the mismatch will cause an error.

Check the data type for the 'High Water Mark' column in the source and update the indexer configuration accordingly. Once verified and updated, reset and rerun the indexer to process the column values.

Error: Access denied to Virtual Network/Firewall rules.

This error typically occurs due to one of the following:

- Firewall restrictions on Azure resources required by your indexer, depending on your configuration. These resources may include: the [data source](#), Azure Storage account (used for [debug sessions](#), [incremental enrichment](#) or [knowledge store](#)), Azure Function (used for [web API custom skills](#)) or AI Services / AI Foundry deployments used during [AI enrichment](#).

- Private endpoint configurations that block access from the indexer to those resources.

Ensure that the indexer has access to your setup components by reviewing your resource configurations to confirm they allow traffic to all required services:

- [Firewall and IP restriction settings](#)
- [Shared private link setup](#)

Error: Credentials provided in the connection string are invalid or have expired.

This error occurs when the Azure AI Search indexer cannot authenticate using the provided connection string or it has issues accessing the storage account to verify the credentials.

[\[\]](#) [Expand table](#)

Possible Cause	Details/Example	Resolution
Expired or rotated key	A connection string contains an outdated key that no longer works.	Go to the resource that is being contacted (for example, Azure Storage or Azure SQL) and copy the latest access keys if using key-based authentication, then update the data source or connection string accordingly.
Managed identity not enabled or access not granted	The AI Search service managed identity is enabled but lacks the required access roles.	- Enable system or user-assigned managed identity on the search Service. - Assign appropriate role(s) to the identity (for example, <code>Storage Blob Data Reader</code> for blob containers). Each data source has its own permission requirements.
Network/firewall blocks identity access	The resource contacted is configured to restrict network access.	Configure network settings to allow Azure AI Search access.
Key authorization has been disabled	Shared key access removed on the source, but the Search service data source configuration still uses key-based authentication.	Use managed identity authentication and ensure role-based permissions are in place. From an Azure Storage perspective, this means that shared key authorization functionality is blocked , either from the storage account itself, or enforced through enterprise-level Azure Policies.

Data sources gallery

Find a data connector from Microsoft or a partner that works with [an indexer](#) to simplify data ingestion into a search index.

! Note

The connectors mentioned in this article represent one method for indexing data in Azure AI Search. You also have the option of developing your own connector using the [Push REST API or An Azure SDK](#).

Generally available data sources by Azure AI Search

Pull in content from other Azure services using indexers and the following data source connectors.

Azure Blob Storage

By [Azure AI Search](#)

Extract blob metadata and content, serialized into JSON documents, and imported into a search index as search documents. Set properties in both data source and indexer definitions to optimize for various blob content types. Change detection is supported automatically.

[More details](#)



Azure Table Storage

By [Azure AI Search](#)

Extract rows from an Azure Table, serialized into JSON documents, and imported into a search index as search documents.

[More details](#)



Azure Data Lake Storage Gen2

By [Azure AI Search](#)

Connect to Azure Storage through Azure Data Lake Storage Gen2 to extract content from a hierarchy of directories and nested subdirectories.

[More details](#)



Azure Cosmos DB for NoSQL

By [Azure AI Search](#)

Connect to Azure Cosmos DB through the SQL API to extract items from a container, serialized into JSON documents, and imported into a search index as search documents. Configure change tracking to refresh the search index with the latest changes in your database.

[More details](#)



Azure SQL Database

By [Azure AI Search](#)

Extract field values from a single table or view, serialized into JSON documents, and imported into a search index as search documents. Configure change tracking to refresh the search index with the latest changes in your database.

[More details](#)



Microsoft OneLake files

By [Azure AI Search](#)

Connect to a OneLake lakehouse to extract supported files content from a hierarchy of directories and nested subdirectories.

[More details](#)



Logic app connectors

Pull in content using [logic app workflows](#) and the following supported data sources. Note that the Logic Apps artifacts mentioned below, they have a pre-built workflow, however, you can use [any connectors listed under Logic Apps](#) that pull data from sources and create your own indexing pipeline workflow that pushes data to [Azure AI Search](#) via a Logic App connector.

SharePoint

By [Logic Apps](#)

SharePoint helps organizations share and collaborate with colleagues, partners, and customers. You can connect to SharePoint in Microsoft 365 or to an on-premises SharePoint 2016 or 2019

farm using the On-Premises Data Gateway to manage documents and list items.

[More details](#)



OneDrive

By [Logic Apps](#)

Connect to OneDrive to manage your files. You can perform various actions such as upload, update, get, and delete on files in OneDrive.

[More details](#)

OneDrive for Business

By [Logic Apps](#)

OneDrive for Business is a cloud storage, file hosting service that allows users to sync files and later access them from a web browser or mobile device. Connect to OneDrive for Business to manage your files. You can perform various actions such as upload, update, get, and delete files.

[More details](#)

Azure File Storage

By [Logic Apps](#)

Microsoft Azure Storage provides a massively scalable, durable, and highly available storage for data on the cloud, and serves as the data storage solution for modern applications. Connect to File Storage to perform various operations such as create, update, get and delete on files in your Azure Storage account.

[More details](#)

Azure Queues

By [Logic Apps](#)

Azure Queue storage provides cloud messaging between application components. Queue storage also supports managing asynchronous tasks and building process work flows.

[More details](#)

Service Bus

By [Logic Apps](#)

Connect to Azure Service Bus to send and receive messages. You can perform actions such as send to queue, send to topic, receive from queue, receive from subscription, etc.

[More details](#)

Preview data sources by Azure AI Search

New data sources are issued as preview features. [Sign up](#) to get started.

Azure Files

By [Azure AI Search](#)

Connect to Azure Storage through Azure Files share to extract content serialized into JSON documents, and imported into a search index as search documents.

[More details](#)



Azure Cosmos DB for Apache Gremlin

By [Azure AI Search](#)

Connect to Azure Cosmos DB for Apache Gremlin to extract items from a container, serialized into JSON documents, and imported into a search index as search documents. Configure change tracking to refresh the search index with the latest changes in your database.

[More details](#)



Azure Cosmos DB for MongoDB

By [Azure AI Search](#)

Connect to Azure Cosmos DB for MongoDB to extract items from a container, serialized into JSON documents, and imported into a search index as search documents. Configure change tracking to refresh the search index with the latest changes in your database.

[More details](#)



SharePoint

By [Azure AI Search](#)

Connect to a SharePoint site and index documents from one or more document libraries, for accounts and search services in the same tenant. Text and normalized images are extracted by default. Optionally, you can configure a skillset for more content transformation and enrichment, or configure change tracking to refresh a search index with new or changed content in SharePoint.

[More details](#)



Azure MySQL

By [Azure AI Search](#)

Connect to MySQL database on Azure to extract rows in a table, serialized into JSON documents, and imported into a search index as search documents. On subsequent runs, assuming High Water Mark change detection policy is configured, the indexer takes all changes, uploads, and delete and reflect those changes in your search index.

[More details](#)



Data sources from our partners

The following Microsoft partners offer custom third-party data connectors. Each partner implements and supports these connectors, which aren't part of Azure AI Search built-in indexers. Before you use a custom connector, review the partner's licensing and usage instructions.

- [BA Insight ↗](#)
- [RheinInsights ↗](#)
- [ServiceNow ↗](#)

Search over Azure Blob Storage content

10/06/2025

Searching across the variety of content types stored in Azure Blob Storage can be a difficult problem to solve, but [Azure AI Search](#) provides deep integration at the content layer, extracting and inferring textual information, which can then be queried in a search index.

In this article, review the basic workflow for extracting content and metadata from blobs and sending it to a [search index](#) in Azure AI Search. The resulting index can be queried using full text search or vector search. Optionally, you can send processed blob content to a [knowledge store](#) for non-search scenarios.

 **Note**

Already familiar with the workflow and composition? [Configure a blob indexer](#) is your next step.

What it means to add search over blob data

Azure AI Search is a standalone search service that supports indexing and query workloads over user-defined indexes that contain your private searchable content hosted in the cloud. Co-locating your searchable content with the query engine in the cloud is necessary for performance, returning results at a speed users have come to expect from search queries.

Azure AI Search integrates with Azure Blob Storage at the indexing layer, importing your blob content as search documents that are indexed into *inverted indexes* and other query structures that support free-form text queries, vector queries, and filter expressions. Because your blob content is indexed into a search index, you can use the full range of query features in Azure AI Search to find information in your blob content.

Inputs are your blobs, in a single container, in Azure Blob Storage. Blobs can be almost any kind of text data. If your blobs contain images, you can add [AI enrichment](#) to create and extract text and features from images.

Output is always an Azure AI Search index, used for fast text search, retrieval, and exploration in client applications. In between is the indexing pipeline architecture itself. The pipeline is based on the *indexer* feature, discussed further on in this article.

Once the index is created and populated, it exists independently of your blob container, but you can rerun indexing operations to refresh your index based on changed documents.

Timestamp information on individual blobs is used for change detection. You can opt for either scheduled execution or on-demand indexing as the refresh mechanism.

Resources used in a blob-search solution

You need Azure AI Search, Azure Blob Storage, and a client. Azure AI Search is typically one of several components in a solution, where your application code issues query API requests and handles the response. You might also write application code to handle indexing, although for proof-of-concept testing and impromptu tasks, it's common to use the Azure portal as the search client.

Within Blob Storage, you'll need a container that provides source content. You can set file inclusion and exclusion criteria, and specify which parts of a blob are indexed in Azure AI Search.

You can start directly in your Storage Account portal page.

1. In the left navigation page under **Data management**, select **Azure AI Search** to select or create a search service.
2. Use an [import wizard](#) to extract and optionally create searchable content from your blobs.

The workflow creates an indexer, data source, index, and option skillset on your Azure AI Search service.

The screenshot shows the Microsoft Azure portal interface for a storage account named 'contoso-storage'. The left sidebar lists various management options like Data management, Storage tasks, and Azure AI Search. The main area is titled 'contoso-storage | Azure AI Search' and contains a form for creating a new search service. The 'Data management' section is selected. The form fields include:

- Select a search service: *Connect to your data
- Add cognitive skills (Optional)
- Customize target index
- Create an indexer
- Data Source: Azure Blob Storage
- Data source name: demo-ds
- Data to extract: Content and metadata
- Parsing mode: Default
- Connection string: DefaultEndpointsProtocol=https;AccountName=heidisteenstor... (with a note to choose an existing connection)
- Managed identity authentication: None (selected)
- Container name: (empty field)
- Blob folder: your/folder/here
- Description: (optional)

At the bottom, there are 'Previous: Select a search service' and 'Next: Add cognitive skills (Optional)' buttons, along with a 'Give feedback' link.

3. Use [Search explorer](#) in the search portal page to query your content.

The wizard is the best place to start, but you'll discover more flexible options when you [configure a blob indexer](#) yourself. You can use a [REST client](#). [Tutorial: Index and search semi-structured data \(JSON blobs\)](#) walks you through the steps of calling the REST API.

How blobs are indexed

By default, most blobs are indexed as a single search document in the index, including blobs with structured content, such as JSON or CSV, which are indexed as a single chunk of text. However, for JSON or CSV documents that have an internal structure (delimiters), you can assign parsing modes to generate individual search documents for each line or element:

- [Indexing JSON blobs](#)
- [Indexing CSV blobs](#)
- [Indexing Markdown blobs](#)
- [Indexing plain text blobs](#)

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding [AI enrichment](#) to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

 **Note**

Azure AI Search imposes [indexer limits](#) on how much text it extracts depending on the pricing tier. A warning will appear in the indexer status response if documents are truncated.

Use a blob indexer for content extraction

An *indexer* is a data-source-aware subservice in Azure AI Search, equipped with internal logic for sampling data, reading and retrieving data and metadata, and serializing data from native formats into JSON documents for subsequent import.

Blobs in Azure Storage are indexed using the [blob indexer](#). You can invoke this indexer by using the [Azure AI Search](#) command in Azure Storage, an [import wizard](#) in the Azure portal, a REST API, or the .NET SDK. In code, you use this indexer by setting the type, and by providing connection information that includes an Azure Storage account along with a blob container.

You can subset your blobs by creating a virtual directory, which you can then pass as a parameter, or by filtering on a file type extension.

An indexer "[cracks a document](#)", opening a blob to inspect content. After connecting to the data source, it's the first step in the pipeline. For blob data, this is where PDF, Office docs, and other content types are detected. Document cracking with text extraction is no charge. If your blobs contain image content, images are ignored unless you [add AI enrichment](#). Standard indexing applies only to text content.

The Azure blob indexer comes with configuration parameters and supports change tracking if the underlying data provides sufficient information. You can learn more about the core functionality in [Index data from Azure Blob Storage](#).

Supported access tiers

Blob storage [access tiers](#) include hot, cool, cold, and archive. Indexers can retrieve blobs on hot, cool, and cold access tiers.

Supported content types

By running a blob indexer over a container, you can extract text and metadata from the following content types with a single query:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCX, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Controlling which blobs are indexed

You can control which blobs are indexed, and which are skipped, by the blob's file type or by setting properties on the blob themselves, causing the indexer to skip over them.

Include specific file extensions by setting `"indexedFileNameExtensions"` to a comma-separated list of file extensions (with a leading dot). Exclude specific file extensions by setting `"excludedFileNameExtensions"` to the extensions that should be skipped. If the same extension is in both lists, it's excluded from indexing.

HTTP

```
PUT /indexers/[indexer name]?api-version=2025-09-01
{
  "parameters" : {
    "configuration" : {
      "indexedFileNameExtensions" : ".pdf, .docx",
      "excludedFileNameExtensions" : ".png, .jpeg"
    }
  }
}
```

Add "skip" metadata the blob

The indexer configuration parameters apply to all blobs in the container or folder. Sometimes, you want to control how *individual blobs* are indexed.

Add the following metadata properties and values to blobs in Blob Storage. When the indexer encounters this property, it skips the blob or its content in the indexing run.

 Expand table

Property name	Property value	Explanation
<code>"AzureSearch_Skip"</code>	<code>"true"</code>	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
<code>"AzureSearch_SkipContent"</code>	<code>"true"</code>	This is equivalent to the <code>"dataToExtract" : "allMetadata"</code> setting described above scoped to a particular blob.

Indexing blob metadata

A common scenario that makes it easy to sort through blobs of any content type is to [index both custom metadata and system properties](#) for each blob. In this way, information for all

blobs is indexed regardless of document type, stored in an index in your search service. Using your new index, you can then proceed to sort, filter, and facet across all Blob storage content.

! Note

Blob Index tags are natively indexed by the Blob storage service and exposed for querying. If your blobs' key/value attributes require indexing and filtering capabilities, Blob Index tags should be used instead of metadata.

To learn more about Blob Index, see [Manage and find data on Azure Blob Storage with Blob Index](#).

Search blob content in a search index

The output of an indexer is a search index, used for interactive exploration using free text and filtered queries in a client app. For initial exploration and verification of content, we recommend starting with [Search Explorer](#) in the Azure portal to examine document structure. In Search explorer, you can use:

- [Simple query syntax](#)
- [Full query syntax](#)
- [Filter expression syntax](#)

A more permanent solution is to gather query inputs and present the response as search results in a client application. The following C# tutorial explains how to build a search application: [Add search to an ASP.NET Core \(MVC\) application](#).

Next steps

- [Upload, download, and list blobs with the Azure portal \(Azure Blob storage\)](#)
- [Set up a blob indexer \(Azure AI Search\)](#)
- [Index large data sets](#)

Index data from Azure Data Lake Storage Gen2

10/09/2025

In this article, learn how to configure an [indexer](#) that imports content from Azure Data Lake Storage (ADLS) Gen2 and makes it searchable in Azure AI Search. Inputs to the indexer are your blobs, in a single container. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements [Create an indexer](#) with information that's specific to indexing from ADLS Gen2. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

For a code sample in C#, see [Index Data Lake Gen2 using Microsoft Entra ID](#) on GitHub.

! Note

ADLS Gen2 supports an [access control model](#) with Azure role-based access control (Azure RBAC) and POSIX-like access control lists (ACLs) at the blob level. Azure AI Search can now recognize document-level permissions in ADLS Gen2 blobs during indexing and transfers those permissions to indexed content in the search index. For more information about ACL ingestion and RBAC scope during indexing, see [Indexing Access Control Lists and Azure Role-Based Access Control scope using Indexers](#).

Prerequisites

- ADLS Gen2 with [hierarchical namespace](#) enabled. ADLS Gen2 is available through Azure Storage. When setting up a storage account, you have the option of enabling [hierarchical namespace](#), organizing files into a hierarchy of directories and nested subdirectories. By enabling a hierarchical namespace, you enable ADLS Gen2.
- [Access tiers](#) for ADLS Gen2 include hot, cool, and archive. Only hot and cool can be accessed by search indexers.
- Blobs containing text. If you have binary data, you can include [AI enrichment](#) for image analysis. Blob content can't exceed the [indexer limits](#) for your search service tier.
- Read permissions on Azure Storage. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles instead, make sure the [search](#)

service managed identity has **Storage Blob Data Reader** permissions.

- Use a [REST client](#) to formulate REST calls similar to the ones shown in this article.

Limitations

- Unlike blob indexers, ADLS Gen2 indexers can't use container-level SAS tokens for enumerating and indexing content from a storage account. This is because the indexer makes a check to determine if the storage account has hierarchical namespaces enabled by calling the [Filesystem - Get properties API](#). For storage accounts where hierarchical namespaces are not enabled, customers are instead recommended to utilize [blob indexers](#) to ensure performant enumeration of blobs.
- If the property `metadata_storage_path` is mapped to be the index key field, blobs are not guaranteed to get reindexed upon a directory rename. If you desire to reindex the blobs that are part of the renamed directories, update the `LastModified` timestamps for all of them.

Supported document formats

The ADLS Gen2 indexer can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Determine which blobs to index

Before you set up indexing, review your source data to determine whether any changes should be made up front. An indexer can index content from one container at a time. By default, all blobs in the container are processed. You have several options for more selective processing:

- Place blobs in a virtual folder. An indexer [data source definition](#) includes a "query" parameter that can take a virtual folder. If you specify a virtual folder, only those blobs in the folder are indexed.
- Include or exclude blobs by file type. The [supported document formats list](#) can help you determine which blobs to exclude. For example, you might want to exclude image or audio files that don't provide searchable text. This capability is controlled through [configuration settings](#) in the indexer.
- Include or exclude arbitrary blobs. If you want to skip a specific blob for whatever reason, you can add the following metadata properties and values to blobs in Blob Storage. When an indexer encounters this property, it skips the blob or its content in the indexing run.

 [Expand table](#)

Property name	Property value	Explanation
"AzureSearch_Skip"	"true"	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
"AzureSearch_SkipContent"	"true"	Skips content and extracts just the metadata. this is equivalent to the <code>"dataToExtract" : "allMetadata"</code> setting described in configuration settings , just scoped to a particular blob.

If you don't set up inclusion or exclusion criteria, the indexer will report an ineligible blob as an error and move on. If enough errors occur, processing might stop. You can specify error tolerance in the indexer [configuration settings](#).

An indexer typically creates one search document per blob, where the text content and metadata are captured as searchable fields in an index. If blobs are whole files, you can potentially parse them into [multiple search documents](#). For example, you can parse rows in a [CSV file](#) to create one search document per row.

Indexing blob metadata

Blob metadata can also be indexed, and that's helpful if you think any of the standard or custom metadata properties will be useful in filters and queries.

User-specified metadata properties are extracted verbatim. To receive the values, you must define field in the search index of type `Edm.String`, with same name as the metadata key of the blob. For example, if a blob has a metadata key of `Sensitivity` with value `High`, you should define a field named `Sensitivity` in your search index and it will be populated with the value `High`.

Standard blob metadata properties can be extracted into similarly named and typed fields, as listed below. The blob indexer automatically creates internal field mappings for these blob metadata properties, converting the original hyphenated name ("metadata-storage-name") to an underscored equivalent name ("metadata_storage_name").

You still have to add the underscored fields to the index definition, but you can omit field mappings because the indexer will make the association automatically.

- **metadata_storage_name** (`Edm.String`) - the file name of the blob. For example, if you have a blob /my-container/my-folder/subfolder/resume.pdf, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the blob, including the storage account. For example, `https://myaccount.blob.core.windows.net/my-container/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the blob. For example, `application/octet-stream`.
- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the blob. Azure AI Search uses this timestamp to identify changed blobs, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - blob size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the blob content, if available.
- **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by [custom skills](#) to get access to the blob. This token shouldn't be stored for later use as it might expire.

Lastly, any metadata properties specific to the document format of the blobs you're indexing can also be represented in the index schema. For more information about content-specific

metadata, see [Content metadata properties](#).

It's important to point out that you don't need to define fields for all of the above properties in your search index - just capture the properties you need for your application.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. [Create or update a data source](#) to set its definition:

JSON

```
{  
  "name" : "my-adlsgen2-datasource",  
  "type" : "adlsgen2",  
  "credentials" : { "connectionString" :  
    "DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=<account key>" },  
  "container" : { "name" : "my-container", "query" : "<optional-virtual-directory-name>" }  
}
```

2. Set "type" to `adlsgen2` (required).

3. Set `credentials` to an Azure Storage connection string. The next section describes the supported formats.

4. Set `container` to the blob container, and use "query" to specify any subfolders.

A data source definition can also include [soft deletion policies](#), if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a blob container using the following connections.

 [Expand table](#)

Full access storage account connection string

```
{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>" }
```

Full access storage account connection string

You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left pane. Make sure to select a full connection string and not just a key.

 Expand table

Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }
```

This connection string doesn't require an account key, but you must have previously configured a search service to [connect using a managed identity](#).

 Expand table

Storage account shared access signature** (SAS) connection string

```
{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=r1;" }
```

The SAS should have the list and read permissions on containers and objects (blobs in this case).

Note

If you use SAS credentials, you will need to update the data source credentials periodically with renewed signatures to prevent their expiration. If SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Add search fields to an index

In a [search index](#), add fields to accept the content and metadata of your Azure blobs.

1. [Create or update an index](#) to define search fields that will store blob content and metadata:

HTTP

```
{
  "name" : "my-search-index",
  "fields": [
```

```

        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false },
        { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_content_type", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true }
    ]
}

```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties.

- `metadata_storage_path` (default) full path to the object or file. The key field ("ID" in this example) will be populated with values from `metadata_storage_path` because it's the default.
- `metadata_storage_name`, usable only if names are unique. If you want this field as the key, move `"key": true` to this field definition.
- A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

Metadata properties often include characters, such as `/` and `-`, that are invalid for document keys. The indexer automatically encodes the key metadata property, with no configuration or field mapping required.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.
4. Add fields for standard metadata properties. The indexer can read custom metadata properties, [standard metadata](#) properties, and [content-specific metadata](#) properties.

Configure and run the ADLS Gen2 indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

You can also specify which parts of a blob to index.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
{  
  "name" : "my-adlsgen2-indexer",  
  "dataSourceName" : "my-adlsgen2-datasource",  
  "targetIndexName" : "my-search-index",  
  "parameters": {  
    "batchSize": null,  
    "maxFailedItems": null,  
    "maxFailedItemsPerBatch": null,  
    "configuration": {  
      "indexedFileNameExtensions" : ".pdf,.docx",  
      "excludedFileNameExtensions" : ".png,.jpeg",  
      "dataToExtract": "contentAndMetadata",  
      "parsingMode": "default"  
    }  
  },  
  "schedule" : { },  
  "fieldMappings" : [ ]  
}
```

2. Set "batchSize" if the default (10 documents) is either under utilizing or overwhelming available resources. Default batch sizes are data source specific. Blob indexing sets batch size at 10 documents in recognition of the larger average document size.
3. Under "configuration", control which blobs are indexed based on file type, or leave unspecified to retrieve all blobs.

For "indexedFileNameExtensions", provide a comma-separated list of file extensions (with a leading dot). Do the same for "excludedFileNameExtensions" to indicate which extensions should be skipped. If the same extension is in both lists, it will be excluded from indexing.

4. Under "configuration", set "dataToExtract" to control which parts of the blobs are indexed:

- "contentAndMetadata" specifies that all metadata and textual content extracted from the blob are indexed. This is the default value.
- "storageMetadata" specifies that only the [standard blob properties and user-specified metadata](#) are indexed.
- "allMetadata" specifies that standard blob properties and any [metadata for found content types](#) are extracted from the blob content and indexed.

5. Under "configuration", set "parsingMode" if blobs should be mapped to [multiple search documents](#), or if they consist of [plain text](#), [JSON documents](#), or [CSV files](#).

6. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

In blob indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens - with underscores in the search index.

7. See [Create an indexer](#) for more information about other properties. For the full list of parameter descriptions, see [Create Indexer \(REST\)](#) in the REST API.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, send a [Get Indexer Status](#) request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status": "running",
    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2024-02-21T00:23:24.957Z",
        "endTime": "2024-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
        "initialTrackingState": null,
        "finalTrackingState": null
    },
    "executionHistory":
```

```

[
  {
    "status": "success",
    "errorMessage": null,
    "startTime": "2024-02-21T00:23:24.957Z",
    "endTime": "2024-02-21T00:36:47.752Z",
    "errors": [],
    "itemsProcessed": 1599501,
    "itemsFailed": 0,
    "initialTrackingState": null,
    "finalTrackingState": null
  },
  ... earlier history items
]
}

```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Handle errors

Errors that commonly occur during indexing include unsupported content types, missing content, or oversized blobs.

By default, the blob indexer stops as soon as it encounters a blob with an unsupported content type (for example, an audio file). You could use the "excludedFileNameExtensions" parameter to skip certain content types. However, you might want indexing to proceed even if errors occur, and then debug individual documents later. For more information about indexer errors, see [Indexer troubleshooting guidance](#) and [Indexer errors and warnings](#).

There are five indexer properties that control the indexer's response when errors occur.

HTTP

```

PUT /indexers/[indexer name]?api-version=2025-09-01
{
  "parameters" : {
    "maxFailedItems" : 10,
    "maxFailedItemsPerBatch" : 10,
    "configuration" : {
      "failOnUnsupportedContentType" : false,
      "failOnUnprocessableDocument" : false,
      "indexStorageMetadataOnlyForOversizedDocuments": false
    }
  }
}

```

Parameter	Valid values	Description
"maxFailedItems"	-1, null or 0, positive integer	Continue indexing if errors happen at any point of processing, either while parsing blobs or while adding documents to an index. Set these properties to the number of acceptable failures. A value of -1 allows processing no matter how many errors occur. Otherwise, the value is a positive integer.
"maxFailedItemsPerBatch"	-1, null or 0, positive integer	Same as above, but used for batch indexing.
"failOnUnsupportedContentType"	true or false	If the indexer is unable to determine the content type, specify whether to continue or fail the job.
"failOnUnprocessableDocument"	true or false	If the indexer is unable to process a document of an otherwise supported content type, specify whether to continue or fail the job.
"indexStorageMetadataOnlyForOversizedDocuments"	true or false	Oversized blobs are treated as errors by default. If you set this parameter to true, the indexer will try to index its metadata even if the content cannot be indexed. For limits on blob size, see service Limits .

Next steps

You can now [run the indexer](#), [monitor status](#), or [schedule indexer execution](#). The following articles apply to indexers that pull content from Azure Storage:

- [Change detection and deletion detection](#)
- [Index large data sets](#)
- [C# Sample: Index Data Lake Gen2 using Microsoft Entra ID](#)

Index data from Azure Blob Storage

10/09/2025

In this article, learn how to configure an [indexer](#) that imports content from Azure Blob Storage and makes it searchable in Azure AI Search. Inputs to the indexer are your blobs, in a single container. Output is a search index with searchable content and metadata stored in individual fields.

To configure and run the indexer, you can use:

- [Search Service REST API](#), any version.
- An Azure SDK package, any version.
- [Import data wizard](#) in the Azure portal.
- [Import data \(new\) wizard](#) in the Azure portal.

This article uses the REST APIs to illustrate each step.

ⓘ Note

Azure AI Search can now ingest RBAC scope during indexing and transfers those permissions to indexed content in the search index. For more information about RBAC scope during indexing, see [Indexing Azure Role-Based Access Control scope using Indexers](#).

Prerequisites

- [Azure Blob Storage](#), Standard performance (general-purpose v2).
- [Access tiers](#) include hot, cool, cold, and archive. Indexers can retrieve blobs on hot, cool, and cold access tiers.
- Blobs providing text content and metadata. If blobs contain binary content or unstructured text, consider adding [AI enrichment](#) for image and natural language processing. Blob content can't exceed the [indexer limits](#) for your search service tier.
- A supported network configuration and data access. At a minimum, you need read permissions in Azure Storage. A storage connection string that includes an access key gives you read access to storage content. If instead you're using Microsoft Entra logins and roles, make sure the [search service's managed identity](#) has [Storage Blob Data Reader](#) permissions.

By default, both search and storage accept requests from public IP addresses. If network security isn't an immediate concern, you can index blob data using just the connection string and read permissions. When you're ready to add network protections, see [Indexer access to content protected by Azure network security features](#) for guidance about data access.

- Use a [REST client](#) to formulate REST calls similar to the ones shown in this article.

Supported tasks

You can use this indexer for the following tasks:

- **Data indexing and incremental indexing:** The indexer can index files and associated metadata from blob containers and folders. It detects new and updated files and metadata through built-in change detection. You can configure data refresh on a schedule or on demand.
- **Deletion detection:** The indexer can [detect deletions through native soft delete or through custom metadata](#).
- **Applied AI through skillsets:** [Skillsets](#) are fully supported by the indexer. This includes key features like [integrated vectorization](#) that adds data chunking and embedding steps.
- **Parsing modes:** The indexer supports [JSON parsing modes](#) if you want to parse JSON arrays or lines into individual search documents. It also supports [Markdown parsing mode](#).
- **Compatibility with other features:** The indexer is designed to work seamlessly with other indexer features, such as [debug sessions](#), [indexer cache for incremental enrichments](#), and [knowledge store](#).

Supported document formats

The blob indexer can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP

- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Determine which blobs to index

Before you set up indexing, review your source data to determine whether any changes should be made up front. An indexer can index content from one container at a time. By default, all blobs in the container are processed. You have several options for more selective processing:

- Place blobs in a virtual folder. An indexer [data source definition](#) includes a "query" parameter that can take a virtual folder. If you specify a virtual folder, only those blobs in the folder are indexed.
- Include or exclude blobs by file type. The [supported document formats list](#) can help you determine which blobs to exclude. For example, you might want to exclude image or audio files that don't provide searchable text. This capability is controlled through [configuration settings](#) in the indexer.
- Include or exclude arbitrary blobs. If you want to skip a specific blob for whatever reason, you can add the following metadata properties and values to blobs in Blob Storage. When an indexer encounters this property, it skips the blob or its content in the indexing run.

[] [Expand table](#)

Property name	Property value	Explanation
"AzureSearch_Skip"	"true"	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
"AzureSearch_SkipContent"	"true"	Skips content and extracts just the metadata. this is equivalent to the "dataToExtract" : "allMetadata" setting described in configuration settings , just scoped to a particular blob.

If you don't set up inclusion or exclusion criteria, the indexer reports an ineligible blob as an error and move on. If enough errors occur, processing might stop. You can specify error tolerance in the indexer [configuration settings](#).

An indexer typically creates one search document per blob, where the text content and metadata are captured as searchable fields in an index. If blobs are whole files, you can potentially parse them into [multiple search documents](#). For example, you can parse rows in a [CSV file](#) to create one search document per row.

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding [AI enrichment](#) to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

Indexing blob metadata

Blob metadata can also be indexed, and that's helpful if you think any of the standard or custom metadata properties are useful in filters and queries.

User-specified metadata properties are extracted verbatim. To receive the values, you must define field in the search index of type `Edm.String`, with same name as the metadata key of the blob. For example, if a blob has a metadata key of `Sensitivity` with value `High`, you should define a field named `Sensitivity` in your search index and it will be populated with the value `High`.

Standard blob metadata properties can be extracted into similarly named and typed fields, as listed below. The blob indexer automatically creates internal field mappings for these blob metadata properties, converting the original hyphenated name ("metadata-storage-name") to an underscored equivalent name ("metadata_storage_name").

You still have to add the underscored fields to the index definition, but you can omit field mappings because the indexer makes the association automatically.

- **metadata_storage_name** (`Edm.String`) - the file name of the blob. For example, if you have a blob /my-container/my-folder/subfolder/resume.pdf, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the blob, including the storage account. For example, `https://myaccount.blob.core.windows.net/my-container/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the blob. For example, `application/octet-stream`.

- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the blob. Azure AI Search uses this timestamp to identify changed blobs, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - blob size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the blob content, if available.
- **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by [custom skills](#) to get access to the blob. This token shouldn't be stored for later use as it might expire.

Lastly, any metadata properties specific to the document format of the blobs you're indexing can also be represented in the index schema. For more information about content-specific metadata, see [Content metadata properties](#).

It's important to point out that you don't need to define fields for all of the above properties in your search index - just capture the properties you need for your application.

Currently, indexing [blob index tags](#) isn't supported by this indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. [Create or update a data source](#) to set its definition:

```
JSON

{
  "name" : "my-blob-datasource",
  "type" : "azureblob",
  "credentials" : { "connectionString" :
  "DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=
  <account key>" },
  "container" : { "name" : "my-container", "query" : "<optional-virtual-
  directory-name>" }
}
```

2. Set "type" to `"azureblob"` (required).

3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.

4. Set "container" to the blob container, and use "query" to specify any subfolders.

A data source definition can also include [soft deletion policies](#), if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a blob container using the following connections.

[\[+\] Expand table](#)

Full access storage account connection string

```
{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }
```

You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left pane. Make sure to select a full connection string and not just a key.

[\[+\] Expand table](#)

Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }
```

This connection string doesn't require an account key, but you must have previously configured a search service to [connect using a managed identity](#).

[\[+\] Expand table](#)

Storage account shared access signature** (SAS) connection string

```
{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=r;" }
```

The SAS should have the list and read permissions on containers and objects (blobs in this case).

[\[+\] Expand table](#)

Container shared access signature

```
{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=r;" }
```

The SAS should have the list and read permissions on the container. For more information, see [Using Shared Access Signatures](#).

! Note

If you use SAS credentials, you will need to update the data source credentials periodically with renewed signatures to prevent their expiration. If SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Add search fields to an index

In a [search index](#), add fields to accept the content and metadata of your Azure blobs.

1. [Create or update an index](#) to define search fields that will store blob content and metadata:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-09-01
{
    "name" : "my-search-index",
    "fields": [
        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false },
        { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_content_type", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true }
    ]
}
```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties.

- `metadata_storage_path` (default) full path to the object or file. The key field ("ID" in this example) will be populated with values from `metadata_storage_path` because it's the default.
- `metadata_storage_name`, usable only if names are unique. If you want this field as the key, move `"key": true` to this field definition.
- A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

Metadata properties often include characters, such as `/` and `-`, which are invalid for document keys. However, the indexer automatically encodes the key metadata property, with no configuration or field mapping required.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.
4. Add fields for standard metadata properties. The indexer can read custom metadata properties, [standard metadata](#) properties, and [content-specific metadata](#) properties.

Configure and run the blob indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors. You can also specify which parts of a blob to index.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
{
  "name" : "my-blob-indexer",
  "dataSourceName" : "my-blob-datasource",
  "targetIndexName" : "my-search-index",
  "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
```

```

    "maxFailedItemsPerBatch": null,
    "configuration": {
        "indexedFileNameExtensions" : ".pdf,.docx",
        "excludedFileNameExtensions" : ".png,.jpeg",
        "dataToExtract": "contentAndMetadata",
        "parsingMode": "default"
    }
},
"schedule" : { },
"fieldMappings" : [ ]
}

```

2. Set `batchSize` if the default (10 documents) is either underutilizing or overwhelming available resources. Default batch sizes are data source specific. Blob indexing sets batch size at 10 documents in recognition of the larger average document size.
3. Under "configuration", control which blobs are indexed based on file type, or leave unspecified to retrieve all blobs.

For `"indexedFileNameExtensions"`, provide a comma-separated list of file extensions (with a leading dot). Do the same for `"excludedFileNameExtensions"` to indicate which extensions should be skipped. If the same extension is in both lists, it will be excluded from indexing.

4. Under "configuration", set "dataToExtract" to control which parts of the blobs are indexed:
 - "contentAndMetadata" specifies that all metadata and textual content extracted from the blob are indexed. This is the default value.
 - "storageMetadata" specifies that only the [standard blob properties and user-specified metadata](#) are indexed.
 - "allMetadata" specifies that standard blob properties and any [metadata for found content types](#) are extracted from the blob content and indexed.
5. Under "configuration", set "parsingMode". The default parsing mode is one search document per blob. If blobs are plain text, you can get better performance by switching to [plain text](#) parsing. If you need more granular parsing that maps blobs to [multiple search documents](#), specify a different mode. One-to-many parsing is supported for blobs consisting of:
 - [JSON documents](#)
 - [CSV files](#)
6. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

In blob indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens - with underscores in the search index.

7. See [Create an indexer](#) for more information about other properties. For the full list of parameter descriptions, see [REST API](#).

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Indexing data from multiple Azure Blob containers to a single index

Keep in mind that an indexer can only index data from a single container. If your requirement is to index data from multiple containers and consolidate it into a single AI Search index, this can be achieved by configuring multiple indexers, all directed to the same index. Please be aware of the [maximum number of indexers available per SKU](#).

To illustrate, let's consider an example of two indexers, pulling data from two distinct data sources, named `my-blob-datasource1` and `my-blob-datasource2`. Each data source points to a separate Azure Blob container, but both direct to the same index named `my-search-index`.

First indexer definition example:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
{
    "name" : "my-blob-indexer1",
    "dataSourceName" : "my-blob-datasource1",
    "targetIndexName" : "my-search-index",
    "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "maxFailedItemsPerBatch": null,
        "configuration": {
            "indexedFileNameExtensions" : ".pdf,.docx",
            "excludedFileNameExtensions" : ".png,.jpeg",
            "dataToExtract": "contentAndMetadata",
            "parsingMode": "default"
        }
    },
    "schedule" : { },
    "fieldMappings" : [ ]
}
```

Second indexer definition that runs in parallel example:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
{
  "name" : "my-blob-indexer2",
  "dataSourceName" : "my-blob-datasource2",
  "targetIndexName" : "my-search-index",
  "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null,
    "configuration": {
      "indexedFileNameExtensions" : ".pdf,.docx",
      "excludedFileNameExtensions" : ".png,.jpeg",
      "dataToExtract": "contentAndMetadata",
      "parsingMode": "default"
    }
  },
  "schedule" : { },
  "fieldMappings" : [ ]
}
```

Check indexer status

To monitor the indexer status and execution history, send a [Get Indexer Status](#) request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-
version=2025-09-01
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
  "status":"running",
  "lastResult": {
    "status":"success",
    "errorMessage":null,
    "startTime":"2022-02-21T00:23:24.957Z",
    "endTime":"2022-02-21T00:36:47.752Z",
    "errors":[],
    "itemsProcessed":1599501,
```

```
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
    },
    "executionHistory":
    [
        {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        ... earlier history items
    ]
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Handle errors

Errors that commonly occur during indexing include unsupported content types, missing content, or oversized blobs.

By default, the blob indexer stops as soon as it encounters a blob with an unsupported content type (for example, an audio file). You could use the "excludedFileNameExtensions" parameter to skip certain content types. However, you might want to indexing to proceed even if errors occur, and then debug individual documents later. For more information about indexer errors, see [Indexer troubleshooting guidance](#) and [Indexer errors and warnings](#).

There are five indexer properties that control the indexer's response when errors occur.

HTTP

```
PUT /indexers/[indexer name]?api-version=2025-09-01
{
    "parameters" : {
        "maxFailedItems" : 10,
        "maxFailedItemsPerBatch" : 10,
        "configuration" : {
            "failOnUnsupportedContentType" : false,
            "failOnUnprocessableDocument" : false,
            "indexStorageMetadataOnlyForOversizedDocuments": false
        }
}
```

```
    }  
}
```

 Expand table

Parameter	Valid values	Description
"maxFailedItems"	-1, null or 0, positive integer	Continue indexing if errors happen at any point of processing, either while parsing blobs or while adding documents to an index. Set these properties to the number of acceptable failures. A value of -1 allows processing no matter how many errors occur. Otherwise, the value is a positive integer.
"maxFailedItemsPerBatch"	-1, null or 0, positive integer	Same as above, but used for batch indexing.
"failOnUnsupportedContentType"	true or false	If the indexer is unable to determine the content type, specify whether to continue or fail the job.
"failOnUnprocessableDocument"	true or false	If the indexer is unable to process a document of an otherwise supported content type, specify whether to continue or fail the job.
"indexStorageMetadataOnlyForOversizedDocuments"	true or false	Oversized blobs are treated as errors by default. If you set this parameter to true, the indexer will try to index its metadata even if the content can't be indexed. For limits on blob size, see service Limits .

See also

- [Change detection and deletion detection](#)
- [Index large data sets](#)
- [Indexer access to content protected by Azure network security features](#)

Index data from Azure Files

05/08/2025

ⓘ Important

Azure Files indexer is currently in public preview under [Supplemental Terms of Use](#). Use a [preview REST API](#) to create the indexer data source.

In this article, learn how to configure an [indexer](#) that imports content from Azure Files and makes it searchable in Azure AI Search. Inputs to the indexer are your files in a single share. Output is a search index with searchable content and metadata stored in individual fields.

To configure and run the indexer, you can use:

- [Search Service preview REST APIs](#), any preview version.
- An Azure SDK package, any version.
- [Import data wizard](#) in the Azure portal.
- [Import and vectorize data wizard](#) in the Azure portal.

Prerequisites

- [Azure Files](#), Transaction Optimized tier.
- An [SMB file share](#) providing the source content. [NFS shares](#) are not supported.
- Files containing text. If you have binary data, you can include [AI enrichment](#) for image analysis.
- Read permissions on Azure Storage. A "full access" connection string includes a key that grants access to the content.
- Use a [REST client](#) to formulate REST calls similar to the ones shown in this article.

Supported tasks

You can use this indexer for the following tasks:

- **Data indexing and incremental indexing:** The indexer can index files and associated metadata from tables. It detects new and updated files and metadata through built-in change detection. You can configure data refresh on a schedule or on demand.
- **Deletion detection:** The indexer can [detect deletions through custom metadata](#).

- **Applied AI through skillsets:** [Skillsets](#) are fully supported by the indexer. This includes key features like [integrated vectorization](#) that adds data chunking and embedding steps.
- **Parsing modes:** The indexer supports [JSON parsing modes](#) if you want to parse JSON arrays or lines into individual search documents. It also supports [Markdown parsing mode](#).
- **Compatibility with other features:** The indexer is designed to work seamlessly with other indexer features, such as [debug sessions](#), [indexer cache for incremental enrichments](#), and [knowledge store](#).

Supported document formats

The Azure Files indexer can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

How Azure Files are indexed

By default, most files are indexed as a single search document in the index, including files with structured content, such as JSON or CSV, which are indexed as a single chunk of text.

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding [AI enrichment](#) to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

You can use 2020-06-30-preview or later for "type": "azurefile". We recommend the latest preview API.

1. [Create a data source](#) to set its definition, using a preview API for "type": "azurefile".

```
HTTP  
  
POST /datasources?api-version=2024-05-01-preview  
{  
    "name" : "my-file-datasource",  
    "type" : "azurefile",  
    "credentials" : { "connectionString" :  
        "DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=<account key>;" },  
    "container" : { "name" : "my-file-share", "query" : "<optional-directory-name>" }  
}
```

2. Set "type" to "azurefile" (required).
3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.
4. Set "container" to the root file share, and use "query" to specify any subfolders.

A data source definition can also include [soft deletion policies](#), if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a file share using the following connections.

[] Expand table

Full access storage account connection string

```
{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }
```

You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left pane. Make sure to select a full connection string and not just a key.

Add search fields to an index

In the [search index](#), add fields to accept the content and metadata of your Azure files.

1. [Create or update an index](#) to define search fields that will store file content and metadata.

HTTP

```
POST /indexes?api-version=2024-07-01
{
    "name" : "my-search-index",
    "fields": [
        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false },
        { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_path", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": true, "sortable": true },
        { "name": "metadata_storage_content_type", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true }
    ]
}
```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties. Metadata properties often include characters, such as / and -, that are invalid for document keys. The indexer automatically encodes the key metadata property, with no configuration or field mapping required.

- `metadata_storage_path` (default) full path to the object or file
- `metadata_storage_name` usable only if names are unique
- A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you

use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.
4. Add fields for standard metadata properties. In file indexing, the standard metadata properties are the same as blob metadata properties. The Azure Files indexer automatically creates internal field mappings for these properties that converts hyphenated property names to underscored property names. You still have to add the fields you want to use the index definition, but you can omit creating field mappings in the data source.

- **metadata_storage_name** (`Edm.String`) - the file name. For example, if you have a file `/my-share/my-folder/subfolder/resume.pdf`, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the file, including the storage account. For example, `https://myaccount.file.core.windows.net/my-share/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the file. For example, `application/octet-stream`.
- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the file. Azure AI Search uses this timestamp to identify changed files, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - file size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the file content, if available.
- **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by [custom skills](#) to get access to the file. This token shouldn't be stored for later use as it might expire.

Configure and run the Azure Files indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
POST /indexers?api-version=2024-07-01
{
    "name" : "my-file-indexer",
    "dataSourceName" : "my-file-datasource",
    "targetIndexName" : "my-search-index",
    "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "maxFailedItemsPerBatch": null,
        "configuration": {
            "indexedFileNameExtensions" : ".pdf,.docx",
            "excludedFileNameExtensions" : ".png,.jpeg"
        }
    },
    "schedule" : { },
    "fieldMappings" : [ ]
}
```

2. In the optional "configuration" section, provide any inclusion or exclusion criteria. If left unspecified, all files in the file share are retrieved.

If both `indexedFileNameExtensions` and `excludedFileNameExtensions` parameters are present, Azure AI Search first looks at `indexedFileNameExtensions`, then at `excludedFileNameExtensions`. If the same file extension is present in both lists, it will be excluded from indexing.

3. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

In file indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens `-` with underscores in the search index.

4. See [Create an indexer](#) for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, send a [Get Indexer Status](#) request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2024-07-01
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status": "running",
    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2022-02-21T00:23:24.957Z",
        "endTime": "2022-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
        "initialTrackingState": null,
        "finalTrackingState": null
    },
    "executionHistory":
    [
        {
            "status": "success",
            "errorMessage": null,
            "startTime": "2022-02-21T00:23:24.957Z",
            "endTime": "2022-02-21T00:36:47.752Z",
            "errors": [],
            "itemsProcessed": 1599501,
            "itemsFailed": 0,
            "initialTrackingState": null,
            "finalTrackingState": null
        },
        ...
        ... earlier history items
    ]
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Next steps

You can now [run the indexer](#), [monitor status](#), or [schedule indexer execution](#). The following articles apply to indexers that pull content from Azure Storage:

- Change detection and deletion detection
- Index large data sets

Index data from Azure Table Storage

10/09/2025

In this article, learn how to configure an [indexer](#) that imports content from Azure Table Storage and makes it searchable in Azure AI Search. Inputs to the indexer are your entities, in a single table. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements [Create an indexer](#) with information that's specific to indexing from Azure Table Storage. It uses the Azure portal and REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Prerequisites

- [Azure Table Storage](#)
- Tables containing text. If you have binary data, consider [AI enrichment](#) for image analysis.
- Read permissions on Azure Storage. A "full access" connection string includes a key that gives access to the content, but if you're using Azure roles, make sure the [search service managed identity](#) has [Storage Table Data Reader](#) permissions.

To work through the examples in this article, you need the Azure portal or a [REST client](#). If you're using Azure portal, make sure that access to all public networks is enabled. Other approaches for creating an Azure Table indexer include Azure SDKs.

Try with sample data

Use these instructions to create a table in Azure Storage for testing purposes.

1. Sign in to the Azure portal, navigate to your storage account, and create a table named *hotels*.
2. [Install Azure Storage Explorer](#).
3. [Download HotelsData_toAzureSearch.csv](#) from GitHub. This file is a subset of the built-in hotels sample dataset. It omits the rooms collection, translated descriptions, and geography coordinates.
4. In Azure Storage Explorer, sign in to Azure, select your subscription, and then select your storage account.

5. Open **Tables** and select *hotels*.
6. Select **Import** on the command bar, and then select the *HotelsData_toAzureSearch.csv* file.
7. Accept the defaults. Select **Import** to load the data.

You should have 50 hotel records in the table with an autogenerated partitionKey, rowKey, and timestamp. You can now use this content for indexing in the Azure portal, REST client, or an Azure SDK.

The Description field provides the most verbose content. You should target this field for full text search and optional vector queries.

Use the Azure portal

You can use either the **Import data** wizard or the **Import data (new)** wizard to automate indexing from an SQL database table or view. The data source configuration is similar for both wizards.

1. [Start the wizard](#).
2. On **Connect to your data**, select or verify that the data source type is either *Azure Table Storage* or that the data selection fields prompt for tables.

The data source name refers to the data source connection object in Azure AI Search. If you use the vector wizard, your data source name is autogenerated using a custom prefix specified at the end of the wizard workflow.

3. Specify the storage account and table name. The query is optional. It's useful if you have specific columns you want to import.
4. Specify an authentication method, either a managed identity or built-in API key. If you don't specify a managed identity connection, the Azure portal uses the key.

If you [configure Azure AI Search to use a managed identity](#), and you create a role assignment on Azure Storage that grants **Reader and Data Access** permissions to the identity, your indexer can connect to table storage using Microsoft Entra ID and roles.

5. For the **Import data (new)** wizard, you can specify options for deletion detection.

Deletion detection requires that you have a preexisting field in the table that can be used as a soft-delete flag. It should be a Boolean field (you could name it *IsDeleted*). Specify `true` as the soft-delete value. In the search index, add a corresponding search field called *IsDeleted* set to retrievable and filterable.

6. Continue with the remaining steps to complete the wizard:

- [Import data wizard](#)
- [Import data \(new\) wizard](#)

Use the REST APIs

This section demonstrates the REST API calls that create a data source, index, and indexer.

Define the data source

The data source definition specifies the source data to index, credentials, and policies for change detection. A data source is an independent resource that can be used by multiple indexers.

1. [Create or update a data source](#) to set its definition:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
{
    "name": "my-table-storage-ds",
    "description": null,
    "type": "azuretable",
    "subtype": null,
    "credentials": {
        "connectionString": "DefaultEndpointsProtocol=https;AccountName=<account name>"
    },
    "container": {
        "name": "my-table-in-azure-storage",
        "query": ""
    },
    "dataChangeDetectionPolicy": null,
    "dataDeletionDetectionPolicy": null,
    "encryptionKey": null,
    "identity": null
}
```

2. Set "type" to `"azuretable"` (required).

3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.

4. Set "container" to the name of the table.

5. Optionally, set "query" to a filter on PartitionKey. Setting this property is a best practice that improves performance. If "query" is null, the indexer executes a full table scan, which can result in poor performance if the tables are large.

A data source definition can also include [soft deletion policies](#), if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a table using the following connections.

[Expand table](#)

Full access storage account connection string

```
{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }
```

You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left pane. Make sure to select a full connection string and not just a key.

[Expand table](#)

Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }
```

This connection string doesn't require an account key, but you must have previously configured a search service to [connect using a managed identity](#).

[Expand table](#)

Storage account shared access signature** (SAS) connection string

```
{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=r;" }
```

The SAS should have the list and read permissions on tables and entities.

[Expand table](#)

Container shared access signature

```
{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=r;" }
```

The SAS should have the list and read permissions on the container. For more information, see [Using Shared Access Signatures](#).

! Note

If you use SAS credentials, you'll need to update the data source credentials periodically with renewed signatures to prevent their expiration. When SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Partition for improved performance

By default, Azure AI Search uses the following internal query filter to keep track of which source entities have been updated since the last run: `Timestamp >= HighWaterMarkValue`. Because Azure tables don't have a secondary index on the `Timestamp` field, this type of query requires a full table scan and is therefore slow for large tables.

To avoid a full scan, you can use table partitions to narrow the scope of each indexer job.

- If your data can naturally be partitioned into several partition ranges, create a data source and a corresponding indexer for each partition range. Each indexer now has to process only a specific partition range, resulting in better query performance. If the data that needs to be indexed has a small number of fixed partitions, even better: each indexer only does a partition scan.

For example, to create a data source for processing a partition range with keys from `000` to `100`, use a query like this: `"container" : { "name" : "my-table", "query" : "PartitionKey ge '000' and PartitionKey lt '100'" }`

- If your data is partitioned by time (for example, if you create a new partition every day or week), consider the following approach:
 - In the data source definition, specify a query similar to the following example: `(PartitionKey ge <TimeStamp>) and (other filters)`.
 - Monitor indexer progress by using [Get Indexer Status API](#), and periodically update the `<TimeStamp>` condition of the query based on the latest successful high-water-mark

value.

- With this approach, if you need to trigger a full reindex, reset the data source query in addition to [resetting the indexer](#).

Add search fields to an index

In a [search index](#), add fields to accept the content and metadata of your table entities.

- [Create or update an index](#) to define search fields that will store content from entities:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-09-01
{
  "name" : "my-search-index",
  "fields": [
    { "name": "Key", "type": "Edm.String", "key": true, "searchable": false },
    { "name": "SomeColumnInMyTable", "type": "Edm.String", "searchable": true }
  ]
}
```

- Create a document key field ("key": true), but allow the indexer to populate it automatically. A table indexer populates the key field with concatenated partition and row keys from the table. For example, if a row's PartitionKey is `1` and RowKey is `1_123`, then the key value is `11_123`. If the partition key is null, just the row key is used.

If you're using the Import data wizard to create the index, the Azure portal infers a "Key" field for the search index and uses an implicit field mapping to connect the source and destination fields. You don't have to add the field yourself, and you don't need to set up a field mapping.

If you're using the REST APIs and you want implicit field mappings, create and name the document key field "Key" in the search index definition as shown in the previous step (`{ "name": "Key", "type": "Edm.String", "key": true, "searchable": false }`). The indexer populates the Key field automatically, with no field mappings required.

If you don't want a field named "Key" in your search index, add an explicit field mapping in the indexer definition with the field name you want, setting the source field to "Key":

JSON

```
"fieldMappings" : [
  {
```

```

        "sourceFieldName" : "Key",
        "targetFieldName" : "MyDocumentKeyFieldName"
    }
]

```

3. Now add any other entity fields that you want in your index. For example, if an entity looks like the following example, your search index should have fields for HotelName, Description, and Category to receive those values.

PartitionKey	RowKey	Timestamp	HotelName	Description	Category
	20	2022-01-14T20:19:10.29...	Travel Resort	The Best Gaming Resort ...	Budget
	23	2022-01-14T20:20:36.88...	Days Hotel	Mix and mingle in the h...	Boutique
	3	2022-01-14T20:21:20.61...	Triple Landscape Hotel	The Hotel stands out for...	Resort and Spa
	31	2022-01-14T20:20:01.02...	Santa Fe Stay	Nestled on six beautiful...	Resort and Spa

Using the same names and compatible [data types](#) minimizes the need for [field mappings](#). When names and types are the same, the indexer can determine the data path automatically.

Configure and run the table indexer

Once you have an index and data source, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```

POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
{
    "name" : "my-table-indexer",
    "dataSourceName" : "my-table-storage-ds",
    "targetIndexName" : "my-search-index",
    "disabled": null,
    "schedule": null,
    "parameters" : {
        "batchSize" : null,
        "maxFailedItems" : null,
        "maxFailedItemsPerBatch" : null,
        "configuration" : { }
    },
    "fieldMappings" : [ ],
    "cache": null,
    "encryptionKey": null
}

```

2. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index. The Target field is the name of the field in the search index.

```
JSON

"fieldMappings" : [
  {
    "sourceFieldName" : "Description",
    "targetFieldName" : "HotelDescription"
  }
]
```

3. See [Create an indexer](#) for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, check the indexer execution history in the Azure portal, or send a [Get Indexer Status](#) REST API request

Portal

1. On the search service page, open **Search management > Indexers**.
2. Select an indexer to access configuration and execution history.
3. Select a specific indexer job to view details, warnings, and errors.

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Next steps

Learn more about how to [run the indexer](#), [monitor status](#), or [schedule indexer execution](#). The following articles apply to indexers that pull content from Azure Storage:

- [Tutorial: Index JSON blobs from Azure Storage](#)
- [Tutorial: Index encrypted blobs in Azure Storage](#)

Change and delete detection using indexers for Azure Storage in Azure AI Search

10/09/2025

After an initial search index is created, you might want subsequent indexer jobs to only pick up new and changed documents. For indexed content that originates from Azure Storage, change detection occurs automatically because indexers keep track of the last update using the built-in timestamps on objects and files in Azure Storage.

Although change detection is a given, deletion detection isn't. An indexer doesn't track object deletion in data sources. To avoid having orphan search documents, you can implement a "soft delete" strategy that results in deleting search documents first, with physical deletion in Azure Storage following as a second step.

There are two ways to implement a soft delete strategy:

- [Native blob soft delete](#), applies to Blob Storage only
- [Soft delete using custom metadata](#)

The deletion detection strategy must be applied from the very first indexer run. If you didn't establish the deletion policy prior to the initial run, any documents that were deleted before the policy was implemented will remain in your index, even if you add the policy to the indexer later and reset it. If this has occurred, it's suggested that you create a new index using a new indexer, ensuring the deletion policy is in place from the beginning.

Prerequisites

- Use an Azure Storage indexer for [Blob Storage](#), [Table Storage](#), [File Storage](#), or [Data Lake Storage Gen2](#)
- Use consistent document keys and file structure. Changing document keys or directory names and paths (applies to ADLS Gen2) breaks the internal tracking information used by indexers to know which content was indexed, and when it was last indexed.

Note

ADLS Gen2 allows directories to be renamed. When a directory is renamed, the timestamps for the blobs in that directory don't get updated. As a result, the indexer won't reindex those blobs. If you need the blobs in a directory to be reindexed after a directory

rename because they now have new URLs, you need to update the `LastModified` timestamp for all the blobs in the directory so that the indexer knows to reindex them during a future run. The virtual directories in Azure Blob Storage can't be changed, so they don't have this issue.

Native blob soft delete

For this deletion detection approach, Azure AI Search depends on the [native blob soft delete](#) feature in Azure Blob Storage to determine whether blobs have transitioned to a soft deleted state. When blobs are detected in this state, a search indexer uses this information to remove the corresponding document from the index.

Requirements for native soft delete

- Blobs must be in an Azure Blob Storage container, including ADLS Gen2 Blob container. The Azure AI Search native blob soft delete policy isn't supported for Azure Files.
- [Enable soft delete for blobs](#).
- Document keys for the documents in your index must be mapped to either be a blob property or blob metadata, such as "metadata_storage_path".
- You can use the [REST API](#), or the indexer Data Source configuration in the Azure portal, to configure support for soft delete.
- [Blob versioning](#) must not be enabled in the storage account. Otherwise, native soft delete isn't supported by design.

Configure native soft delete

In Blob storage, when enabling soft delete per the requirements, set the retention policy to a value that's much higher than your indexer interval schedule. If there's an issue running the indexer, or if you have a large number of documents to index, there's plenty of time for the indexer to eventually process the soft deleted blobs. Azure AI Search indexers will only delete a document from the index if it processes the blob while it's in a soft deleted state.

In Azure AI Search, set a native blob soft deletion detection policy on the data source. You can do this either from the Azure portal or by using the [REST API](#). The following instructions explain how to set the delete detection policy in Azure portal or through REST APIs.

Azure portal

1. Sign in to the [Azure portal](#).
2. On the Azure AI Search service Overview page, go to **New Data Source**, a visual editor for specifying a data source definition.

The following screenshot shows where you can find this feature in the Azure portal.

The screenshot shows the Azure AI Search service Overview page for a search service named "azure-cognitive-search-service". The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Settings. The main content area has tabs for Get Started, Usage, Monitoring, Indexes, Indexers, and Data sources. The "Data sources" tab is selected, and a red box highlights the "+ New Data Source" button. Below it, there's a table with one row labeled "my-blob-data-source".

3. On the **New Data Source** form, fill out the required fields, select the **Track deletions** checkbox and choose **Native blob soft delete**. Then hit **Save** to enable the feature on Data Source creation.

The screenshot shows the "New Data Source" configuration form. The "Settings" tab is selected. The "Data Source" dropdown is set to "Azure Blob Storage". The "Name" field contains "my-blob-data-source". The "Connection string" field contains "DefaultEndpointsProtocol=https;AccountName=[accountName];AccountKey=[accountKey]". Under "Managed identity authentication", "None" is selected. The "Container name" field is "mycontainer". The "Blob folder" field is "your/folder/here". The "Track deletions" checkbox is checked (highlighted by a red box). Under "Data deletion detection policy", the "Native blob soft delete" radio button is selected (highlighted by a red box). The "Description" field is "(optional)".

Reindex undeleted blobs using native soft delete policies

If you restore a soft deleted blob in Blob storage, the indexer won't always reindex it. This is because the indexer uses the blob's `LastModified` timestamp to determine whether indexing is needed. When a soft deleted blob is undeleted, its `LastModified` timestamp doesn't get updated, so if the indexer has already processed blobs with more recent `LastModified` timestamps, it won't reindex the undeleted blob.

To make sure that an undeleted blob is reindexed, you'll need to update the blob's `LastModified` timestamp. One way to do this is by resaving the metadata of that blob. You don't need to change the metadata, but resaving the metadata will update the blob's `LastModified` timestamp so that the indexer knows to pick it up.

Soft delete strategy using custom metadata

This method uses custom metadata to indicate whether a search document should be removed from the index. It requires two separate actions: deleting the search document from the index, followed by file deletion in Azure Storage.

This feature is generally available.

There are steps to follow in both Azure Storage and Azure AI Search, but there are no other feature dependencies.

1. In Azure Storage, add a custom metadata key-value pair to the file to indicate the file is flagged for deletion. For example, you could name the property "IsDeleted", set to false. When you want to delete the file, change it to true.
2. In Azure AI Search, edit the data source definition to include a "dataDeletionDetectionPolicy" property. For example, the following policy considers a file to be deleted if it has a metadata property `IsDeleted` with the value `true`:

HTTP

```
PUT https://[service name].search.windows.net/datasources/file-datasource?  
api-version=2025-09-01  
{  
    "name" : "file-datasource",  
    "type" : "azurefile",  
    "credentials" : { "connectionString" : "<your storage connection string>" },  
    "container" : { "name" : "my-share", "query" : null },  
    "dataDeletionDetectionPolicy" : {  
        "@odata.type"  
        :"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
        "softDeleteColumnName" : "IsDeleted",  
        "softDeleteMarkerValue" : "true"
```

```
    }  
}
```

3. Run the indexer. Once the indexer has processed the file and deleted the document from the search index, you can then delete the physical file in Azure Storage.

Reindex undeleted blobs and files

You can reverse a soft-delete if the original source file still physically exists in Azure Storage.

1. Change the `"softDeleteMarkerValue" : "false"` on the blob or file in Azure Storage.
2. Check the blob or file's `LastModified` timestamp to make it's newer than the last indexer run. You can force an update to the current date and time by resaving the existing metadata.
3. Run the indexer.

Next steps

- [Indexers in Azure AI Search](#)
- [How to configure a blob indexer](#)
- [Blob indexing overview](#)

Index data from Azure Cosmos DB for NoSQL for queries in Azure AI Search

10/09/2025

In this article, learn how to configure an [indexer](#) that imports content from [Azure Cosmos DB for NoSQL](#) and makes it searchable in Azure AI Search.

This article supplements [Create an indexer](#) with information that's specific to Cosmos DB. It uses the Azure portal and REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that [Azure Cosmos DB indexing](#) and [Azure AI Search indexing](#) are different operations. Indexing in Azure AI Search creates and loads a search index on your search service.

Prerequisites

- An [Azure Cosmos DB account, database, container, and items](#). Use the same region for both Azure AI Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An [automatic indexing policy](#) on the Azure Cosmos DB collection, set to [Consistent](#). This is the default configuration. Lazy indexing isn't recommended and can result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using identities (Microsoft Entra ID), make sure the [search service managed identity](#) is assigned as both [Cosmos DB Account Reader Role](#) and [Cosmos DB Built-in Data Reader Role](#).

To work through the examples in this article, you need the Azure portal or a [REST client](#). If you're using Azure portal, make sure that access to all public networks is enabled. Other approaches for creating a Cosmos DB indexer include Azure SDKs.

Try with sample data

Use these instructions to create a container and database in Cosmos DB for testing purposes.

1. [Download HotelsData_toCosmosDB.json](#) ↗ from GitHub to create a container in Cosmos DB that contains a subset of the sample hotels data set.

- Sign in to the Azure portal and [create an account, database, and container](#) on Cosmos DB.
- In Cosmos DB, select **Data Explorer** for the new container, provide the following values.

 Expand table

Property	Value
Database	Create new
Database ID	hotelsdb
Share throughput across containers	Don't select
Container ID	hotels
Partition key	/HotelId
Container throughput (autoscale)	Autoscale
Container Max RU/s	1000

- In **Data Explorer**, expand *hotelsdb* and *hotels*, and then select **Items**.
- Select **Upload Item** and then select *HotelsData_toCosmosDB.json* file that you downloaded from GitHub.
- Right-click **Items** and select **New SQL query**. The default query is `SELECT * FROM c`.
- Select **Execute query** to run the query and view results. You should have 50 hotel documents.

Now that you have a container, you can use the Azure portal, REST client, or an Azure SDK to index your data.

The Description field provides the most verbose content. You should target this field for full text search and optional vector queries.

Use the Azure portal

You can use either the **Import data** wizard or the **Import data (new)** wizard to automate indexing from an SQL database table or view. The data source configuration is similar for both wizards.

- [Start the wizard](#).

2. On **Connect to your data**, select or verify that the data source type is either *Azure Cosmos DB* or a *NoSQL account*.

The data source name refers to the data source connection object in Azure AI Search. If you use the vector wizard, your data source name is autogenerated using a custom prefix specified at the end of the wizard workflow.

3. Specify the database name and collection. The query is optional. It's useful if you have hierarchical data and you want to import a specific slice.

4. Specify an authentication method, either a managed identity or built-in API key. If you don't specify a managed identity connection, the Azure portal uses the key.

If you [configure Azure AI Search to use a managed identity](#), and you create a [role assignment on Cosmos DB](#) that grants **Cosmos DB Account Reader** and **Cosmos DB Built-in Data Reader** permissions to the identity, your indexer can connect to Cosmos DB using Microsoft Entra ID and roles.

5. For the **Import data (new)** wizard, you can specify options for change and deletion tracking.

[Change detection](#) is supported by default through a `_ts` field (timestamp). If you upload content using the approach described in [Try with sample data](#), the collection is created with a `_ts` field.

[Deletion detection](#) requires that you have a preexisting top-level field in the collection that can be used as a soft-delete flag. It should be a Boolean field (you could name it `IsDeleted`). Specify `true` as the soft-deleted value. In the search index, add a corresponding search field called `IsDeleted` set to retrievable and filterable.

6. Continue with the remaining steps to complete the wizard:

- [Import data wizard](#)
- [Import data \(new\) wizard](#)

Use the REST APIs

This section demonstrates the REST API calls that create a data source, index, and indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is an independent resource that can be used by multiple

indexers.

1. [Create or update a data source](#) to set its definition:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: [Search service admin key]
{
  "name": "[my-cosmosdb-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name]"
  },
  "container": {
    "name": "[my-cosmos-db-collection]",
    "query": null
  },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

2. Set "type" to `"cosmosdb"` (required). If you're using an older Search API version 2017-11-11, the syntax for "type" is `"documentdb"`. Otherwise, for 2019-05-06 and later, use `"cosmosdb"`.
3. Set "credentials" to a connection string. The next section describes the supported formats.
4. Set "container" to the collection. The "name" property is required and it specifies the ID of the database collection to be indexed. The "query" property is optional. Use it to [flatten an arbitrary JSON document](#) into a flat schema that Azure AI Search can index.
5. [Set "dataChangeDetectionPolicy"](#) if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.
6. [Set "dataDeletionDetectionPolicy"](#) if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections.

Avoid port numbers in the endpoint URL. If you include the port number, the connection fails.

[] [Expand table](#)

Full access connection string

```
{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>" }
```

You can get the connection string from the Azure Cosmos DB account page in the Azure portal by selecting **Keys** in the left pane. Make sure to select a full connection string and not just a key.

[] [Expand table](#)

(Modern approach) Managed identity connection string for NoSQL accounts

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind]);/(IdentityAuthType=AccessToken)" }
```

This connection string, supported only for Azure Cosmos DB for NoSQL accounts, ensures that the search service will never use account keys (even in the background) when attempting to access data from Cosmos DB. This is recommended, as it works even if the NoSQL account has account keys disabled. For more information, see [Setting up an indexer connection to an Azure Cosmos DB database using a managed identity](#)

[] [Expand table](#)

(Legacy approach) Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind]);/(IdentityAuthType=AccountKey)" }
```

This connection string doesn't require an account key to be specified directly, but the search service will utilize the managed identity to fetch the account keys in the background. Though this is supported for all Cosmos DB account types, it isn't recommended for the NoSQL account type. Such a connection string won't work if account keys are disabled for the Cosmos DB account. If the `IdentityAuthType` property is omitted, the search service will still default to fetching the account key in the background. For connections targeting the [SQL API](#), you can omit `ApiKind` from the connection string. For more information about `ApiKind`, `IdentityAuthType` see [Setting up an indexer connection to an Azure Cosmos DB database using a managed identity](#)

Using queries to shape indexed data

In the "query" property under "container", you can specify a SQL query to flatten nested properties or arrays, project JSON properties, and filter the data to be indexed.

Example document:

HTTP

```
{  
    "userId": 10001,  
    "contact": {  
        "firstName": "andy",  
        "lastName": "hoh"  
    },  
    "company": "microsoft",  
    "tags": ["azure", "cosmosdb", "search"]  
}
```

Filter query:

SQL

```
SELECT * FROM c WHERE c.company = "microsoft" and c._ts >= @HighWaterMark ORDER BY c._ts
```

Flattening query:

SQL

```
SELECT c.id, c.userId, c.contact.firstName, c.contact.lastName, c.company, c._ts  
FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

Projection query:

SQL

```
SELECT VALUE { "id":c.id, "Name":c.contact.firstName, "Company":c.company,  
"_ts":c._ts } FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

Array flattening query:

SQL

```
SELECT c.id, c.userId, tag, c._ts FROM c JOIN tag IN c.tags WHERE c._ts >=  
@HighWaterMark ORDER BY c._ts
```

Unsupported queries (DISTINCT and GROUP BY)

Queries using the [DISTINCT keyword](#) or [GROUP BY clause](#) aren't supported. Azure AI Search relies on [SQL query pagination](#) to fully enumerate the results of the query. Neither the DISTINCT keyword or GROUP BY clauses are compatible with the [continuation tokens](#) used to paginate results.

Examples of unsupported queries:

SQL

```
SELECT DISTINCT c.id, c.userId, c._ts FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts

SELECT DISTINCT VALUE c.name FROM c ORDER BY c.name

SELECT TOP 4 COUNT(1) AS foodGroupCount, f.foodGroup FROM Food f GROUP BY f.foodGroup
```

Although Azure Cosmos DB has a workaround to support [SQL query pagination with the DISTINCT keyword by using the ORDER BY clause](#), it isn't compatible with Azure AI Search. The query returns a single JSON value, whereas Azure AI Search expects a JSON object.

SQL

```
-- The following query returns a single JSON value and isn't supported by Azure AI Search
SELECT DISTINCT VALUE c.name FROM c ORDER BY c.name
```

Add search fields to an index

In a [search index](#), add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with source data. For content in Azure Cosmos DB, your search index schema should correspond to the [Azure Cosmos DB items](#) in your data source.

1. [Create or update an index](#) to define search fields that store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-09-01
Content-Type: application/json
api-key: [Search service admin key]
{
    "name": "mysearchindex",
    "fields": [{
```

```

        "name": "rid",
        "type": "Edm.String",
        "key": true,
        "searchable": false
    },
    {
        "name": "description",
        "type": "Edm.String",
        "filterable": false,
        "searchable": true,
        "sortable": false,
        "facetable": false,
        "suggestions": true
    }
]
}

```

2. Create a document key field ("key": true). For partitioned collections, the default document key is the Azure Cosmos DB `_rid` property, which Azure AI Search automatically renames to `rid` because field names can't start with an underscore character. Also, Azure Cosmos DB `_rid` values contain characters that are invalid in Azure AI Search keys. For this reason, the `_rid` values are Base64 encoded.
3. Create more fields for more searchable content. See [Create an index](#) for details.

Mapping data types

[] Expand table

JSON data types	Azure AI Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB for NoSQL indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-cosmosdb-indexer]",
    "dataSourceName" : "[my-cosmosdb-ds]",
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {}
    },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.
3. See [Create an indexer](#) for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, check the indexer execution history in the Azure portal, or send a [Get Indexer Status](#) REST API request

Portal

1. On the search service page, open **Search management > Indexers**.

2. Select an indexer to access configuration and execution history.
3. Select a specific indexer job to view details, warnings, and errors.

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the [HighWaterMarkChangeDetectionPolicy](#) using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a [data source definition](#) with a change detection policy:

HTTP

```
"dataChangeDetectionPolicy": {  
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
    "highWaterMarkColumnName": "_ts"  
},
```

ⓘ Note

When you assign a `null` value to a field in your Azure Cosmos DB, the AI Search indexer is unable to distinguish between `null` and a missing field value. Therefore, if a field in the index is empty, it will not be substituted with a `null` value, even if that modification was made in your database.

Incremental indexing and custom queries

If you're using a [custom query to retrieve documents](#), make sure the query orders the results by the `_ts` column. This enables periodic check-pointing that Azure AI Search uses to provide incremental progress in the presence of failures.

In some cases, even if your query contains an `ORDER BY [collection alias]._ts` clause, Azure AI Search might not infer that the query is ordered by the `_ts`. You can tell Azure AI Search that results are ordered by setting the `assumeOrderByHighWaterMarkColumn` configuration property.

To specify this hint, [create or update your indexer definition](#) as follows:

```
HTTP

{
    ... other indexer definition properties
    "parameters" : {
        "configuration" : { "assumeOrderByHighWaterMarkColumn" : true } }
}
```

Indexing deleted documents

When rows are deleted from the collection, you normally want to delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

```
HTTP

"dataDeletionDetectionPolicy": {
    "@odata.type" :
    "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName" : "the property that specifies whether a document was
    deleted",
    "softDeleteMarkerValue" : "the value that identifies a document as deleted"
}
```

If you're using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The `softDeleteColumnName` must be a top-level field in the index. Using nested fields within complex data types as the `softDeleteColumnName` isn't supported.

The following example creates a data source with a soft-deletion policy:

```
HTTP

POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: [Search service admin key]
```

```
{
  "name": "[my-cosmosdb-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name]"
  },
  "container": { "name": "[my-cosmos-collection]" },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName": "isDeleted",
    "softDeleteMarkerValue": "true"
  }
}
```

Use .NET

For data accessed through the SQL API protocol, you can use the .NET SDK to automate with indexers. We recommend that you review the previous REST API sections to learn concepts, workflow, and requirements. You can then refer to following .NET API reference documentation to implement a JSON indexer in managed code:

- [azure.search.documents.indexes.models.searchindexerdatasourceconnection](#)
- [azure.search.documents.indexes.models.searchindexerdatasourcetype](#)
- [azure.search.documents.indexes.models.searchindex](#)
- [azure.search.documents.indexes.models.searchindexer](#)

Next steps

You can now control how you [run the indexer](#), [monitor status](#), or [schedule indexer execution](#). The following articles apply to indexers that pull content from Azure Cosmos DB:

- [Set up an indexer connection to an Azure Cosmos DB database using a managed identity](#)
- [Index large data sets](#)
- [Indexer access to content protected by Azure network security features](#)

Index data from Azure Cosmos DB for MongoDB for queries in Azure AI Search

10/09/2025

Important

MongoDB API support is currently in public preview under [supplemental Terms of Use](#). Currently, there is no SDK support.

In this article, learn how to configure an [indexer](#) that imports content from [Azure Cosmos DB for MongoDB](#) and makes it searchable in Azure AI Search.

This article supplements [Create an indexer](#) with information that's specific to Cosmos DB. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that [Azure Cosmos DB indexing](#) and [Azure AI Search indexing](#) are different operations. Indexing in Azure AI Search creates and loads a search index on your search service.

Prerequisites

- Register for the preview [to provide scenario feedback](#). You can access the feature automatically after form submission.
- An [Azure Cosmos DB account, database, collection, and documents](#). Use the same region for both Azure AI Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An [automatic indexing policy](#) on the Azure Cosmos DB collection, set to [Consistent](#). This is the default configuration. Lazy indexing isn't recommended and may result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the [search service managed identity](#) has [Cosmos DB Account Reader Role](#) permissions.
- A [REST client](#) to create the data source, index, and indexer.

Limitations

These are the limitations of this feature:

- Custom queries aren't supported for specifying the data set.
- The column name `_ts` is a reserved word. If you need this field, consider alternative solutions for populating an index.
- The MongoDB attribute `$ref` is a reserved word. If you need this in your MongoDB collection, consider alternative solutions for populating an index.

As an alternative to this connector, if your scenario has any of those requirements, you could use the [Push API/SDK](#) or consider [Azure Data Factory](#) with an [Azure AI Search index](#) as the sink.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

For this call, specify a [preview REST API version](#). You can use 2020-06-30-preview or later to create a data source that connects via the MongoDB API. We recommend the latest preview REST API.

1. [Create or update a data source](#) to set its definition:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [Search service admin key]
{
  "name": "[my-cosmosdb-mongodb-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name];ApiKind=MongoDb;"
  },
  "container": {
    "name": "[cosmos-db-collection]",
    "query": null
  },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMark": {
      "start": "2024-01-01T00:00:00Z",
      "end": "2024-01-01T23:59:59Z"
    }
  }
}
```

```
        "highWaterMarkColumnName": "_ts"  
    },  
    "dataDeletionDetectionPolicy": null,  
    "encryptionKey": null,  
    "identity": null  
}
```

2. Set "type" to "cosmosdb" (required).
3. Set "credentials" to a connection string. The next section describes the supported formats.
4. Set "container" to the collection. The "name" property is required and it specifies the ID of the database collection to be indexed. For Azure Cosmos DB for MongoDB, "query" isn't supported.
5. [Set "dataChangeDetectionPolicy"](#) if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.
6. [Set "dataDeletionDetectionPolicy"](#) if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections. For connections that target the [MongoDB API](#), be sure to include "ApiKind" in the connection string.

Avoid port numbers in the endpoint URL. If you include the port number, the connection will fail.

 [Expand table](#)

Full access connection string

```
{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account  
name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database  
id>;ApiKind=MongoDb" }
```

You can get the *Cosmos DB auth key* from the Azure Cosmos DB account page in the Azure portal by selecting **Connection String** in the left pane. Make sure to copy **Primary Password** and replace *Cosmos DB auth key* value with it.

 [Expand table](#)

Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind]);;" }
```

This connection string doesn't require an account key, but you must have previously configured a search service to [connect using a managed identity](#) and created a role assignment that grants **Cosmos DB Account Reader Role** permissions. See [Setting up an indexer connection to an Azure Cosmos DB database using a managed identity](#) for more information.

Add search fields to an index

In a [search index](#), add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with source data. For content in Azure Cosmos DB, your search index schema should correspond to the [Azure Cosmos DB items](#) in your data source.

1. [Create or update an index](#) to define search fields that will store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "mysearchindex",
    "fields": [
        {
            "name": "doc_id",
            "type": "Edm.String",
            "key": true,
            "retrievable": true,
            "searchable": false
        },
        {
            "name": "description",
            "type": "Edm.String",
            "filterable": false,
            "searchable": true,
            "sortable": false,
            "facetable": false,
            "suggestions": true
        }
    ]
}
```

2. Create a document key field ("key": true). For a search index based on a MongoDB collection, the document key can be "doc_id", "rid", or some other string field that

contains unique values. As long as field names and data types are the same on both sides, no field mappings are required.

- "doc_id" represents "_id" for the object identifier. If you specify a field of "doc_id" in the index, the indexer populates it with the values of the object identifier.
- "rid" is a system property in Azure Cosmos DB. If you specify a field of "rid" in the index, the indexer populates it with the base64-encoded value of the "rid" property.
- For any other field, your search field should have the same name as defined in the collection.

3. Create additional fields for more searchable content. See [Create an index](#) for details.

Mapping data types

[] [Expand table](#)

JSON data type	Azure AI Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB for MongoDB indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-cosmosdb-indexer]",
    "dataSourceName" : "[my-cosmosdb-mongodb-ds]",
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {}
    },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.
3. See [Create an indexer](#) for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, send a [Get Indexer Status](#) request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status": "running",
```

```

    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2022-02-21T00:23:24.957Z",
        "endTime": "2022-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
        "initialTrackingState": null,
        "finalTrackingState": null
    },
    "executionHistory": [
        {
            "status": "success",
            "errorMessage": null,
            "startTime": "2022-02-21T00:23:24.957Z",
            "endTime": "2022-02-21T00:36:47.752Z",
            "errors": [],
            "itemsProcessed": 1599501,
            "itemsFailed": 0,
            "initialTrackingState": null,
            "finalTrackingState": null
        },
        ...
        ... earlier history items
    ]
}

```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the [HighWaterMarkChangeDetectionPolicy](#) using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a [data source definition](#) with a change detection policy:

HTTP

```
"dataChangeDetectionPolicy": {  
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
    "highWaterMarkColumnName": "_ts"  
},
```

Indexing deleted documents

When rows are deleted from the collection, you normally want to delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

HTTP

```
"dataDeletionDetectionPolicy": {  
    "@odata.type" :  
    "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
    "softDeleteColumnName" : "the property that specifies whether a document was  
deleted",  
    "softDeleteMarkerValue" : "the value that identifies a document as deleted"  
}
```

If you're using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The following example creates a data source with a soft-deletion policy:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2024-05-01-  
preview  
Content-Type: application/json  
api-key: [Search service admin key]  
  
{  
    "name": ["my-cosmosdb-mongodb-ds"],  
    "type": "cosmosdb",  
    "credentials": {  
        "connectionString": "AccountEndpoint=https://[cosmos-account-  
name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-  
database-name];ApiKind=MongoDB"  
    },  
    "container": { "name": "[my-cosmos-collection]" },  
    "dataChangeDetectionPolicy": {  
        "@odata.type":  
        "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
        "highWaterMarkColumnName": "_ts"
```

```
 },
  "dataDeletionDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionPolicy",
    "softDeleteColumnName": "isDeleted",
    "softDeleteMarkerValue": "true"
  }
}
```

Next steps

You can now control how you [run the indexer](#), [monitor status](#), or [schedule indexer execution](#).

The following articles apply to indexers that pull content from Azure Cosmos DB:

- [Set up an indexer connection to an Azure Cosmos DB database using a managed identity](#)
- [Index large data sets](#)
- [Indexer access to content protected by Azure network security features](#)

Index data from Azure Cosmos DB for Apache Gremlin for queries in Azure AI Search

10/09/2025

Important

The Azure Cosmos DB for Apache Gremlin indexer is currently in public preview under [Supplemental Terms of Use](#). Currently, there is no SDK support.

In this article, learn how to configure an [indexer](#) that imports content from [Azure Cosmos DB for Apache Gremlin](#) and makes it searchable in Azure AI Search.

This article supplements [Create an indexer](#) with information that's specific to Cosmos DB. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that [Azure Cosmos DB indexing](#) and [Azure AI Search indexing](#) are different operations. Indexing in Azure AI Search creates and loads a search index on your search service.

Prerequisites

- Register for the preview [to provide scenario feedback](#). You can access the feature automatically after form submission.
- An [Azure Cosmos DB account, database, container, and items](#). Use the same region for both Azure AI Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An [automatic indexing policy](#) on the Azure Cosmos DB collection, set to [Consistent](#). This is the default configuration. Lazy indexing isn't recommended and may result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the [search service managed identity](#) has [Cosmos DB Account Reader Role](#) permissions.
- A [REST client](#) to create the data source, index, and indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

For this call, specify a [preview REST API version](#) to create a data source that connects via an Azure Cosmos DB for Apache Gremlin. You can use 2021-04-01-preview or later. We recommend the latest preview API.

1. [Create or update a data source](#) to set its definition:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [Search service admin key]
{
  "name": "[my-cosmosdb-gremlin-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name];ApiKind=Gremlin;"
  },
  "container": {
    "name": "[cosmos-db-collection]",
    "query": "g.V()"
  },
  "dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

2. Set "type" to `"cosmosdb"` (required).
3. Set "credentials" to a connection string. The next section describes the supported formats.
4. Set "container" to the collection. The "name" property is required and it specifies the ID of the graph.

The "query" property is optional. By default the Azure AI Search indexer for Azure Cosmos DB for Apache Gremlin makes every vertex in your graph a document in the index. Edges will be ignored. The query default is `g.V()`. Alternatively, you could set the query to only index the edges. To index the edges, set the query to `g.E()`.

5. Set "`dataChangeDetectionPolicy`" if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs. Incremental progress will be enabled by default using `_ts` as the high water mark column.
6. Set "`dataDeletionDetectionPolicy`" if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections. For connections that target [Azure Cosmos DB for Apache Gremlin](#), be sure to include "ApiKind" in the connection string.

Avoid port numbers in the endpoint URL. If you include the port number, the connection will fail.

 Expand table

Full access connection string

```
{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>;ApiKind=MongoDb" }
```

You can get the connection string from the Azure Cosmos DB account page in Azure portal by selecting **Keys** in the left pane. Make sure to select a full connection string and not just a key.

 Expand table

Managed identity connection string

```
{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind]);;" }
```

This connection string doesn't require an account key, but you must have previously configured a search service to [connect using a managed identity](#) and created a role assignment that grants **Cosmos DB Account Reader Role** permissions. See [Setting up an indexer connection to an Azure Cosmos DB database using a managed identity](#) for more information.

Add search fields to an index

In a [search index](#), add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with your graph. For content in Azure Cosmos DB, your search index schema should correspond to the [Azure Cosmos DB items](#) in your data source.

1. [Create or update an index](#) to define search fields that will store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [Search service admin key]
{
  "name": "mysearchindex",
  "fields": [
    {
      "name": "rid",
      "type": "Edm.String",
      "facetable": false,
      "filterable": false,
      "key": true,
      " retrievable": true,
      "searchable": true,
      "sortable": false,
      "analyzer": "standard.lucene",
      "indexAnalyzer": null,
      "searchAnalyzer": null,
      "synonymMaps": [],
      "fields": []
    },
    {
      "name": "label",
      "type": "Edm.String",
      "searchable": true,
      "filterable": false,
      " retrievable": true,
      "sortable": false,
      "facetable": false,
      "key": false,
      "indexAnalyzer": null,
      "searchAnalyzer": null,
      "analyzer": "standard.lucene",
      "synonymMaps": []
    }
  ]
}
```

2. Create a document key field ("key": true). For partitioned collections, the default document key is the Azure Cosmos DB `_rid` property, which Azure AI Search automatically renames to `rid` because field names can't start with an underscore character. Also, Azure Cosmos DB `_rid` values contain characters that are invalid in Azure AI Search keys. For this reason, the `_rid` values are Base64 encoded.

3. Create additional fields for more searchable content. See [Create an index](#) for details.

Mapping data types

 [Expand table](#)

JSON data type	Azure AI Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-cosmosdb-indexer]",
    "dataSourceName" : "[my-cosmosdb-gremlin-ds]".
```

```
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {}
    },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.
3. See [Create an indexer](#) for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

To monitor the indexer status and execution history, send a [Get Indexer Status](#) request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status": "running",
    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2022-02-21T00:23:24.957Z",
        "endTime": "2022-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
```

```

        "initialTrackingState":null,
        "finalTrackingState":null
    },
    "executionHistory":
    [
        {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        ...
        ... earlier history items
    ]
}

```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the [HighWaterMarkChangeDetectionPolicy](#) using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a [data source definition](#) with a change detection policy:

HTTP

```

"dataChangeDetectionPolicy": {
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
},

```

Indexing deleted documents

When graph data is deleted, you might want to delete its corresponding document from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items and delete the full document from the index. The data deletion detection policy isn't meant to delete partial document information. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

HTTP

```
"dataDeletionDetectionPolicy": {
    "@odata.type" : "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName" : "the property that specifies whether a document was deleted",
    "softDeleteMarkerValue" : "the value that identifies a document as deleted"
}
```

The following example creates a data source with a soft-deletion policy:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2024-05-01-preview
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "[my-cosmosdb-gremlin-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name];ApiKind=Gremlin"
    },
    "container": { "name": "[my-cosmos-collection]" },
    "dataChangeDetectionPolicy": {
        "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "`_ts`"
    },
    "dataDeletionDetectionPolicy": {
        "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName": "isDeleted",
        "softDeleteMarkerValue": "true"
    }
}
```

Even if you enable deletion detection policy, deleting complex (`Edm.ComplexType`) fields from the index is not supported. This policy requires that the 'active' column in the Gremlin database to be of type integer, string or boolean.

Mapping graph data to fields in a search index

The Azure Cosmos DB for Apache Gremlin indexer will automatically map a couple pieces of graph data:

1. The indexer will map `_rid` to an `rid` field in the index if it exists, and Base64 encode it.
2. The indexer will map `_id` to an `id` field in the index if it exists.
3. When querying your Azure Cosmos DB database using the Azure Cosmos DB for Apache Gremlin you may notice that the JSON output for each property has an `id` and a `value`. The indexer will automatically map the properties `value` into a field in your search index that has the same name as the property if it exists. In the following example, 450 would be mapped to a `pages` field in the search index.

HTTP

```
{  
    "id": "Cookbook",  
    "label": "book",  
    "type": "vertex",  
    "properties": {  
        "pages": [  
            {  
                "id": "48cf6285-a145-42c8-a0aa-d39079277b71",  
                "value": "450"  
            }  
        ]  
    }  
}
```

You may find that you need to use [Output Field Mappings](#) in order to map your query output to the fields in your index. You'll likely want to use Output Field Mappings instead of [Field Mappings](#) since the custom query will likely have complex data.

For example, let's say that your query produces this output:

JSON

```
[  
    {  
        "vertex": {  
            "label": "book",  
            "type": "vertex",  
            "properties": {  
                "name": "Cookbook",  
                "pages": "450"  
            }  
        }  
    }  
]
```

```
"id": "Cookbook",
"label": "book",
"type": "vertex",
"properties": {
    "pages": [
        {
            "id": "48cf6085-a211-42d8-a8ea-d38642987a71",
            "value": "450"
        }
    ],
    "written_by": [
        {
            "yearStarted": "2017"
        }
    ]
}
```

If you would like to map the value of `pages` in the JSON above to a `totalpages` field in your index, you can add the following [Output Field Mapping](#) to your indexer definition:

JSON

```
... // rest of indexer definition
"outputFieldMappings": [
    {
        "sourceFieldName": "/document/vertex/pages",
        "targetFieldName": "totalpages"
    }
]
```

Notice how the Output Field Mapping starts with `/document` and does not include a reference to the properties key in the JSON. This is because the indexer puts each document under the `/document` node when ingesting the graph data and the indexer also automatically allows you to reference the value of `pages` by simple referencing `pages` instead of having to reference the first object in the array of `pages`.

Next steps

- To learn more about Azure Cosmos DB for Apache Gremlin, see the [Introduction to Azure Cosmos DB: Azure Cosmos DB for Apache Gremlin](#).
- For more information about Azure AI Search scenarios and pricing, see the [Search service page on azure.microsoft.com](#).

- To learn about network configuration for indexers, see the [Indexer access to content protected by Azure network security features](#).

Index data from Azure Database for MySQL Flexible Server

Important

MySQL support is currently in public preview under [Supplemental Terms of Use](#). We recommend the latest preview API. There is currently no portal support.

In this article, learn how to configure an [indexer](#) that imports content from Azure Database for MySQL and makes it searchable in Azure AI Search. Inputs to the indexer are rows from a single table or view. Output is a search index with searchable content in individual fields.

This article supplements [Create an indexer](#) with information that's specific to indexing from Azure Database for MySQL Flexible Server. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

When configured to include a high water mark and soft deletion, the indexer takes all changes, uploads, and deletes for your MySQL database. It reflects these changes in your search index. Data extraction occurs when you submit the Create Indexer request.

Prerequisites

- [Register for the preview](#) to provide scenario feedback. You can access the feature automatically after form submission.
- [Azure Database for MySQL Flexible Server](#) and sample data. Data must reside in a table or view. A primary key is required. If you're using a view, it must have a [high water mark column](#).
- Read permissions. A *full access* connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the [search service managed identity](#) has **Reader** permissions on MySQL.
- A [REST client](#) to create the data source, index, and indexer.

You can also use the [Azure SDK for .NET](#). You can't use the Azure portal for indexer creation, but you can manage indexers and data sources once they're created.

Preview limitations

Currently, change tracking and deletion detection aren't working if the date or timestamp is uniform for all rows. This limitation is a known issue to be addressed in an update to the preview. Until this issue is addressed, don't add a skillset to the MySQL indexer.

The preview doesn't support geometry types and blobs.

As noted, there's no portal support for indexer creation, but a MySQL indexer and data source can be managed in the Azure portal once they exist. For example, you can edit the definitions, and reset, run, or schedule the indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. The data source is defined as an independent resource so that it can be used by multiple indexers.

[Create or Update Data Source](#) specifies the definition. Be sure to use a preview REST API when creating the data source.

HTTP

```
{  
  "name" : "hotel-mysql-ds",  
  "description" : "[Description of MySQL data source]",  
  "type" : "mysql",  
  "credentials" : {  
    "connectionString" :  
      "Server=[MySQLServerName].MySQL.database.azure.com; Port=3306; Database=  
      [DatabaseName]; Uid=[UserName]; Pwd=[Password]; SslMode=Preferred;"  
  },  
  "container" : {  
    "name" : "[TableName]"  
  },  
  "dataChangeDetectionPolicy" : {  
    "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
    "highWaterMarkColumnName": "[HighWaterMarkColumn]"  
  }  
}
```

Key points:

- Set `type` to `"mysql"` (required).
- Set `credentials` to an ADO.NET connection string. You can find connection strings in Azure portal, on the [Connection strings](#) page for MySQL.
- Set `container` to the name of the table.

- Set [dataChangeDetectionPolicy](#) if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.
- Set [dataDeletionDetectionPolicy](#) if you want to remove search documents from a search index when the source item is deleted.

! Note

For the container name property, the value is restricted to only allow letters, numbers, underscores (_), dots (.), single dashes (-), and square brackets ([])

Create an index

[Create or Update Index](#) specifies the index schema:

HTTP

```
{  
  "name" : "hotels-mysql-ix",  
  "fields": [  
    { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },  
    { "name": "HotelName", "type": "Edm.String", "searchable": true,  
    "filterable": false },  
    { "name": "Category", "type": "Edm.String", "searchable": false,  
    "filterable": true, "sortable": true },  
    { "name": "City", "type": "Edm.String", "searchable": false, "filterable":  
true, "sortable": true },  
    { "name": "Description", "type": "Edm.String", "searchable": false,  
    "filterable": false, "sortable": false }  
  ]  
}
```

If the primary key in the source table matches the document key (in this case, "ID"), the indexer imports the primary key as the document key.

Mapping data types

The following table maps the MySQL database to Azure AI Search equivalents. For more information, see [Supported data types \(Azure AI Search\)](#).

! Note

The preview does not support geometry types and blobs.

MySQL data types	Azure AI Search field types
<code>bool, boolean</code>	Edm.Boolean, Edm.String
<code>tinyint, smallint, mediumint, int, integer, year</code>	Edm.Int32, Edm.Int64, Edm.String
<code>bigint</code>	Edm.Int64, Edm.String
<code>float, double, real</code>	Edm.Double, Edm.String
<code>date, datetime, timestamp</code>	Edm.DateTimeOffset, Edm.String
<code>char, varchar, tinytext, mediumtext, text, longtext, enum, set, time</code>	Edm.String
unsigned numerical data, serial, decimal, dec, bit, blob, binary, geometry	N/A

Configure and run the MySQL indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

[Create or update an indexer](#) by giving it a name and referencing the data source and target index:

HTTP

```
{
  "name" : "hotels-mysql-idxr",
  "dataSourceName" : "hotels-mysql-ds",
  "targetIndexName" : "hotels-mysql-ix",
  "disabled": null,
  "schedule": null,
  "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null,
    "base64EncodeKeys": null,
    "configuration": { }
  },
  "fieldMappings" : [ ],
  "encryptionKey": null
}
```

Key points:

- Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.
- An indexer runs automatically when it's created. You can prevent it from running by setting `disabled` to `true`. To control indexer execution, [run an indexer on demand](#) or [put it on a schedule](#).

Check indexer status

Send a [Get Indexer Status](#) request to monitor indexer execution:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status": "running",
    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2024-02-21T00:23:24.957Z",
        "endTime": "2024-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
        "initialTrackingState": null,
        "finalTrackingState": null
    },
    "executionHistory": [
        {
            "status": "success",
            "errorMessage": null,
            "startTime": "2024-02-21T00:23:24.957Z",
            "endTime": "2024-02-21T00:36:47.752Z",
            "errors": [],
            "itemsProcessed": 1599501,
            "itemsFailed": 0,
            "initialTrackingState": null,
            "finalTrackingState": null
        },
        ...
    ]
}
```

```
    ... earlier history items  
]  
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed rows

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed rows in your database.

To enable incremental indexing, set the `dataChangeDetectionPolicy` property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Database for MySQL indexers, the only supported policy is the [HighWaterMarkChangeDetectionPolicy](#).

An indexer's change detection policy relies on having a *high water mark* column that captures the row version, or the date and time when a row was last updated. It's often a `DATE`, `DATETIME`, or `TIMESTAMP` column at a granularity sufficient for meeting the requirements of a high water mark column.

In your MySQL database, the high water mark column must meet the following requirements:

- All data inserts must specify a value for the column.
- All updates to an item also change the value of the column.
- The value of this column increases with each insert or update.
- Queries with the following `WHERE` and `ORDER BY` clauses can be executed efficiently: `WHERE [High Water Mark Column] > [Current High Water Mark Value] ORDER BY [High Water Mark Column]`

The following example shows a [data source definition](#) with a change detection policy:

HTTP

```
{  
  "name" : "[Data source name]",  
  "type" : "mysql",  
  "credentials" : { "connectionString" : "[connection string]" },  
  "container" : { "name" : "[table or view name]" },  
  "dataChangeDetectionPolicy" : {  
    "@odata.type" :  
      "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
    "highWaterMarkColumn" : "[High Water Mark Column Name]" }  
}
```

```
        "highWaterMarkColumnName" : "[last_updated column name]"
    }
}
```

Important

If you're using a view, you must set a high water mark policy in your indexer data source.

If the source table does not have an index on the high water mark column, queries used by the MySQL indexer might time out. In particular, the `ORDER BY [High Water Mark Column]` clause requires an index to run efficiently when the table contains many rows.

Indexing deleted rows

When rows are deleted from the table or view, you normally want to delete those rows from the search index as well. However, if the rows are physically removed from the table, an indexer has no way to infer the presence of records that no longer exist. The solution is to use a *soft-delete* technique to logically delete rows without removing them from the table. Add a column to your table or view and mark rows as deleted using that column.

Given a column that provides deletion state, an indexer can be configured to remove any search documents for which deletion state is set to `true`. The configuration property that supports this behavior is a data deletion detection policy, which is specified in the [data source definition](#) as follows:

HTTP

```
{  
    ...,  
    "dataDeletionDetectionPolicy" : {  
        "@odata.type" : "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
        "softDeleteColumnName" : "[a column name]",  
        "softDeleteMarkerValue" : "[the value that indicates that a row is deleted]"  
    }  
}
```

The `softDeleteMarkerValue` must be a string. For example, if you have an integer column where deleted rows are marked with the value 1, use `"1"`. If you have a `BIT` column where deleted rows are marked with the Boolean true value, use the string literal `True` or `true` (the case doesn't matter).

Next steps

You can now [run the indexer](#), [monitor status](#), or [schedule indexer execution](#). The following articles apply to indexers that pull content from Azure MySQL:

- [Index large data sets](#)
- [Indexer access to content protected by Azure network security features](#)

Last updated on 10/09/2025

Index data from Azure SQL Database

In this article, learn how to configure an [indexer](#) that imports content from Azure SQL Database or an Azure SQL managed instance and makes it searchable in Azure AI Search.

This article supplements [Create an indexer](#) with information that's specific to Azure SQL. It uses the Azure portal and REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

This article also provides:

- A description of the [change detection policies](#) supported by the Azure SQL indexer so that you can set up incremental indexing.
- A [frequently-asked-questions \(FAQ\) section](#) for answers to questions about feature compatibility.

! Note

Real-time data synchronization isn't possible with an indexer. An indexer can reindex your table at most every five minutes. If data updates need to be reflected in the index sooner, we recommend [pushing updated rows directly](#).

Prerequisites

- An [Azure SQL database](#) or a [SQL Managed Instance with a public endpoint](#).
- A single table or view.

Use a table if your data is large or if you need incremental indexing using SQL's native change detection capabilities ([SQL integrated change tracking](#)) to reflect new, changed, and deleted rows in the search index.

Use a view if you need to consolidate data from multiple tables. Large views aren't ideal for SQL indexer. A workaround is to create a new table just for ingestion into your Azure AI Search index. If you choose to go with a view, you can use [High Water Mark](#) for change detection, but must use a workaround for deletion detection.

- Primary key must be single-valued. On a table, it must also be non-clustered for full SQL integrated change tracking.

- Read permissions. Azure AI Search supports SQL Server authentication, where the user name and password are provided on the connection string. Alternatively, you can [set up a managed identity and use Azure roles](#) with membership in **SQL Server Contributor** or **SQL DB Contributor** roles.

To work through the examples in this article, you need the Azure portal or a [REST client](#). If you're using Azure portal, make sure that access to all public networks is enabled in the Azure SQL firewall and that the client has access via an inbound rule. For a REST client that runs locally, configure the SQL Server firewall to allow inbound access from your device IP address. Other approaches for creating an Azure SQL indexer include Azure SDKs.

Try with sample data

Use these instructions to create and load a table in Azure SQL Database for testing purposes.

1. [Download hotels-azure-sql.sql](#) ↗ from GitHub to create a table on Azure SQL Database that contains a subset of the sample hotels data set.
2. Sign in to the Azure portal and [create an Azure SQL database and database server](#). Consider configuring both SQL Server authentication and Microsoft Entra ID authentication. If you don't have permissions to configure roles on Azure, you can use SQL authentication as a workaround.
3. Configure the server firewall to all inbound requests from your local device.
4. On your Azure SQL database, select **Query editor (preview)** and then select **New Query**.
5. Paste in and then run the T-SQL script that creates the hotels table. A non-clustered primary key is a requirement for SQL integrated change tracking.

Transact-SQL

```
CREATE TABLE tbl_hotels
(
    Id TINYINT PRIMARY KEY NONCLUSTERED,
    Modified DateTime NULL DEFAULT '0000-00-00 00:00:00',
    IsDeleted TINYINT,
    HotelName VARCHAR(40),
    Category VARCHAR(20),
    City VARCHAR(30),
    State VARCHAR(4),
    Description VARCHAR(500)
);
```

6. Paste in and then run the T-SQL script that inserts records.

Transact-SQL

```
-- Insert rows
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (1, CURRENT_TIMESTAMP, 0, 'Stay-Kay City Hotel', 'Boutique', 'New York', 'NY', 'This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of Americas most attractive and cosmopolitan cities.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (10, CURRENT_TIMESTAMP, 0, 'Countryside Hotel', 'Extended-Stay', 'Durham', 'NC', 'Save up to 50% off traditional hotels. Free WiFi, great location near downtown, full kitchen, washer & dryer, 24\7 support, bowling alley, fitness center and more.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (11, CURRENT_TIMESTAMP, 0, 'Royal Cottage Resort', 'Extended-Stay', 'Bothell', 'WA', 'Your home away from home. Brand new fully equipped premium rooms, fast WiFi, full kitchen, washer & dryer, fitness center. Inner courtyard includes water features and outdoor seating. All units include fireplaces and small outdoor balconies. Pets accepted.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (12, CURRENT_TIMESTAMP, 0, 'Winter Panorama Resort', 'Resort and Spa', 'Wilsonville', 'OR', 'Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, group exercise classes and outdoor hockey are available year-round, plus numerous options for shopping as well as great spa services. Newly-renovated with large rooms, free 24-hr airport shuttle & a new restaurant. Rooms\suites offer mini-fridges & 49-inch HDTVs.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (13, CURRENT_TIMESTAMP, 0, 'Luxury Lion Resort', 'Luxury', 'St. Louis', 'MO', 'Unmatched Luxury. Visit our downtown hotel to indulge in luxury accommodations. Moments from the stadium and transportation hubs, we feature the best in convenience and comfort.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (14, CURRENT_TIMESTAMP, 0, 'Twin Vortex Hotel', 'Luxury', 'Dallas', 'TX', 'New experience in the making. Be the first to experience the luxury of the Twin Vortex. Reserve one of our newly-renovated guest rooms today.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (15, CURRENT_TIMESTAMP, 0, 'By the Market Hotel', 'Budget', 'New York', 'NY', 'Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (16, CURRENT_TIMESTAMP, 0, 'Double Sanctuary Resort', 'Resort and Spa', 'Seattle', 'WA', '5 Star Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Traveler magazine. Free WiFi, Flexible check in\out, Fitness Center & espresso in room.');
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (17, CURRENT_TIMESTAMP, 0, 'City Skyline Antiquity Hotel', 'Boutique', 'New York', 'NY', 'In vogue since 1888, the Antiquity Hotel takes you back to bygone era. From the crystal chandeliers that adorn the Green Room, to the arched ceilings of the Grand Hall, the elegance of old New York beckons. Elevate Your Experience. Upgrade to a premiere city skyline view for
```

```
less, where old world charm combines with dramatic views of the city, local cathedral and midtown.'');
```

```
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (18, CURRENT_TIMESTAMP, 0, 'Ocean Water Resort & Spa', 'Luxury', 'Tampa', 'FL', 'New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near the pier, rooftop pool, waterfront dining & more.');
```

```
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (19, CURRENT_TIMESTAMP, 0, 'Economy Universe Motel', 'Budget', 'Redmond', 'WA', 'Local, family-run hotel in bustling downtown Redmond. We are a pet-friendly establishment, near expansive Marymoor park, haven to pet owners, joggers, and sports enthusiasts. Close to the highway and just a short drive away from major cities.');
```

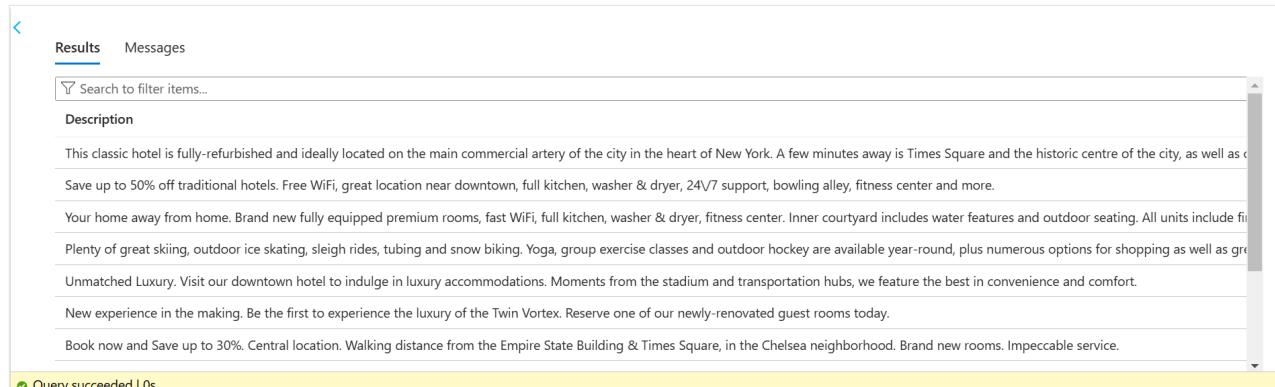
```
INSERT INTO tbl_hotels (Id, Modified, IsDeleted, HotelName, Category, City, State, Description) VALUES (20, CURRENT_TIMESTAMP, 0, 'Delete Me Hotel', 'Unknown', 'Nowhere', 'XX', 'Test-case row for change detection and delete detection . For change detection, modify any value, and then re-run the indexer. For soft-delete, change IsDelete from zero to a one, and then re-run the indexer.');
```

7. Run a query to confirm the upload.

Transact-SQL

```
SELECT Description FROM tbl_hotels;
```

You should see results similar to the following screenshot.



The screenshot shows a SQL query results window. The 'Results' tab is selected. A search bar at the top says 'Search to filter items...'. Below it is a table with a single row. The table has two columns: 'Description'. The content of the 'Description' column is:

```
This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as central business district. Save up to 50% off traditional hotels. Free WiFi, great location near downtown, full kitchen, washer & dryer, 24/7 support, bowling alley, fitness center and more. Your home away from home. Brand new fully equipped premium rooms, fast WiFi, full kitchen, washer & dryer, fitness center. Inner courtyard includes water features and outdoor seating. All units include free WiFi. Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, group exercise classes and outdoor hockey are available year-round, plus numerous options for shopping as well as great restaurants. Unmatched Luxury. Visit our downtown hotel to indulge in luxury accommodations. Moments from the stadium and transportation hubs, we feature the best in convenience and comfort. New experience in the making. Be the first to experience the luxury of the Twin Vortex. Reserve one of our newly-renovated guest rooms today. Book now and Save up to 30%. Central location. Walking distance from the Empire State Building & Times Square, in the Chelsea neighborhood. Brand new rooms. Impeccable service.
```

At the bottom of the results pane, there is a yellow status bar that says 'Query succeeded | 0s'.

The Description field provides the most verbose content. You should target this field for full text search and optional vectorization.

Now that you have a database table, you can use the Azure portal, REST client, or an Azure SDK to index your data.

Tip

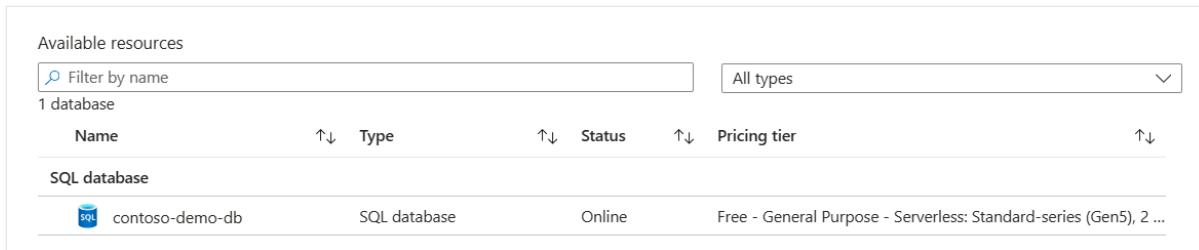
Another resource that provides sample content and code can be found on [Azure-Samples/SQL-AI-samples](#).

Set up the indexer pipeline

In this step, specify the data source, index, and indexer.

Azure portal

1. Make sure your SQL database is active and not paused due to inactivity. In the Azure portal, navigate to the database server page and verify the database status is *online*. You can run a query on any table to activate the database.



The screenshot shows the 'Available resources' blade in the Azure portal. It includes a search bar labeled 'Filter by name' and a dropdown menu set to 'All types'. Below this, there is a table with the following data:

Name	Type	Status	Pricing tier
contoso-demo-db	SQL database	Online	Free - General Purpose - Serverless: Standard-series (Gen5), 2 ...

2. Make sure you have a table or view that meets the requirements for indexers and change detection.

First, you can only pull from a single table or view. We recommend tables because they support SQL integrated change tracking policy, which detects new, updated, and deleted rows. A high water mark policy doesn't support row deletion and is harder to implement.

Second, the primary key must be a single value (compound keys aren't supported) and non-clustered.

3. Switch to your search service and create a data source. Under **Search management > Data sources**, select **Add data source**:
 - a. For data source type, choose *Azure SQL Database*.
 - b. Provide a name for the data source object on Azure AI Search.
 - c. Use the dropdowns to select the subscription, account type, server, database, table or view, schema, and table name.
 - d. For change tracking we recommend **SQL Integrated Change Tracking Policy**.
 - e. For authentication, we recommend connecting with a **managed identity**. Your search service must have **SQL Server Contributor** or **SQL DB Contributor** role membership on the database.
 - f. Select **Create** to create the data source.

Home >

Add data source

Data Source

Data Source	Azure SQL Database
Name *	hotels-ds
Subscription *	my-demo-azure-subscription
Azure SQL account type	SQL databases
Server *	my-demo-sql-server
Database *	hotels-db
Table or View	Table
Schema *	dbo
Table name *	tbl_hotels

 Enable deletion tracking Track changes (1) SQL Integrated Change Tracking Policy High Water Mark Change Detection policy

Select an authentication option

 SQL server authentication Authenticate using managed identity. [Learn more](#)Managed identity type (1)

System-assigned

Advanced options

Encryption

[Microsoft-managed keys](#)

4. Use an [import wizard](#) to create the index and indexer.

- a. On the [Overview](#) page, select **Import data** or **Import data (new)**.
- b. Select the data source you just created.
- c. Skip the step for adding AI enrichments.
- d. Name the index, set the key to your primary key in the table, attribute all fields as **Retrievable** and **Searchable**, and optionally add **Filterable** and **Sortable** for short strings or numeric values.
- e. Name the indexer and finish the wizard to create the necessary objects.

Check indexer status

To monitor the indexer status and execution history, check the indexer execution history in the Azure portal, or send a [Get Indexer Status](#) REST API request

Portal

1. On the search service page, open **Search management > Indexers**.
2. Select an indexer to access configuration and execution history.
3. Select a specific indexer job to view details, warnings, and errors.

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new, changed, and deleted rows

If your SQL database supports [change tracking](#), a search indexer can pick up just the new and updated content on subsequent indexer runs.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your table or view.

For Azure SQL indexers, there are two change detection policies:

- "SqlIntegratedChangeTrackingPolicy" (applies to tables only)
- "HighWaterMarkChangeDetectionPolicy" (works for views)

SQL integrated change tracking policy

We recommend using "SqlIntegratedChangeTrackingPolicy" for its efficiency and its ability to identify deleted rows.

Database requirements:

- Azure SQL Database or SQL Managed Instance. SQL Server 2016 or later if you're using an Azure VM.
- Database must have [change tracking enabled](#)
- Tables only (no views).

- Tables can't be clustered. To meet this requirement, drop the clustered index and recreate it as non-clustered index. This workaround often degrades performance. Duplicating content in a second table that's dedicated to indexer processing can be a helpful mitigation.
- Tables can't be empty. If you use TRUNCATE TABLE to clear rows, a reset and rerun of the indexer won't remove the corresponding search documents. To remove orphaned search documents, you must [index them with a delete action](#).
- Primary key can't be a compound key (containing more than one column).
- Primary key must be non-clustered if you want deletion detection.

Change detection policies are added to data source definitions. To use this policy, edit the data source definition in the Azure portal, or use REST to update your data source like this:

HTTP

```
POST https://myservice.search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: admin-key
{
  "name" : "myazuresqldatasource",
  "type" : "azuresql",
  "credentials" : { "connectionString" : "connection string" },
  "container" : { "name" : "table name" },
  "dataChangeDetectionPolicy" : {
    "@odata.type" :
    "#Microsoft.Azure.Search.SqlIntegratedChangeTrackingPolicy"
  }
}
```

When using SQL integrated change tracking policy, don't specify a separate data deletion detection policy. The SQL integrated change tracking policy has built-in support for identifying deleted rows. However, for the deleted rows to be detected automatically, the document key in your search index must be the same as the primary key in the SQL table, and the primary key must be non-clustered.

High water mark change detection policy

This change detection policy relies on a "high water mark" column in your table or view that captures the version or time when a row was last updated. If you're using a view, you must use a high water mark policy.

The high water mark column must meet the following requirements:

- All inserts specify a value for the column.
- All updates to an item also change the value of the column.

- The value of this column increases with each insert or update.
- Queries with the following WHERE and ORDER BY clauses can be executed efficiently:

```
WHERE [High Water Mark Column] > [Current High Water Mark Value] ORDER BY [High  
Water Mark Column]
```

!*Note*

We strongly recommend using the `rowversion` data type for the high water mark column. If any other data type is used, change tracking isn't guaranteed to capture all changes in the presence of transactions executing concurrently with an indexer query. When using `rowversion` in a configuration with read-only replicas, you must point the indexer at the primary replica. Only a primary replica can be used for data sync scenarios.

Change detection policies are added to data source definitions. To use this policy, create or update your data source like this:

HTTP

```
POST https://myservice.search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: admin-key
{
    "name" : "myazuresqldatasource",
    "type" : "azuresql",
    "credentials" : { "connectionString" : "connection string" },
    "container" : { "name" : "table or view name" },
    "dataChangeDetectionPolicy" : {
        "@odata.type" :
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName" : "[a rowversion or last_updated column name]"
    }
}
```

!*Note*

If the source table doesn't have an index on the high water mark column, queries used by the SQL indexer may time out. In particular, the `ORDER BY [High Water Mark Column]` clause requires an index to run efficiently when the table contains many rows.

convertHighWaterMarkToRowVersion

If you're using a `rowversion` data type for the high water mark column, consider setting the `convertHighWaterMarkToRowVersion` property in indexer configuration. Setting this property to

true results in the following behaviors:

- Uses the rowversion data type for the high water mark column in the indexer SQL query. Using the correct data type improves indexer query performance.
- Subtracts one from the rowversion value before the indexer query runs. Views with one-to-many joins might have rows with duplicate rowversion values. Subtracting one ensures the indexer query doesn't miss these rows.

To enable this property, create or update the indexer with the following configuration:

HTTP

```
{  
    ... other indexer definition properties  
    "parameters" : {  
        "configuration" : { "convertHighWaterMarkToRowVersion" : true } }  
}
```

queryTimeout

If you encounter timeout errors, set the `queryTimeout` indexer configuration setting to a value higher than the default 5-minute timeout. For example, to set the timeout to 10 minutes, create or update the indexer with the following configuration:

HTTP

```
{  
    ... other indexer definition properties  
    "parameters" : {  
        "configuration" : { "queryTimeout" : "00:10:00" } }  
}
```

disableOrderByHighWaterMarkColumn

You can also disable the `ORDER BY [High Water Mark Column]` clause. However, this isn't recommended because if the indexer execution is interrupted by an error, the indexer has to reprocess all rows if it runs later, even if the indexer has already processed almost all the rows at the time it was interrupted. To disable the `ORDER BY` clause, use the `disableOrderByHighWaterMarkColumn` setting in the indexer definition:

HTTP

```
{  
    ... other indexer definition properties
```

```
"parameters" : {  
    "configuration" : { "disableOrderByHighWaterMarkColumn" : true } }  
}
```

Soft delete column deletion detection policy

When rows are deleted from the source table, you probably want to delete those rows from the search index as well. If you use the SQL integrated change tracking policy, this is taken care of for you. However, the high water mark change tracking policy doesn't help you with deleted rows. What to do?

If the rows are physically removed from the table, Azure AI Search has no way to infer the presence of records that no longer exist. However, you can use the "soft-delete" technique to logically delete rows without removing them from the table. Add a column to your table or view and mark rows as deleted using that column.

When using the soft-delete technique, you can specify the soft delete policy as follows when creating or updating the data source:

HTTP

```
{  
    ...,  
    "dataDeletionDetectionPolicy" : {  
        "@odata.type" : "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
        "softDeleteColumnName" : "[a column name]",  
        "softDeleteMarkerValue" : "[the value that indicates that a row is  
deleted]"  
    }  
}
```

The **softDeleteMarkerValue** must be a string in the JSON representation of your data source. Use the string representation of your actual value. For example, if you have an integer column where deleted rows are marked with the value 1, use `"1"`. If you have a BIT column where deleted rows are marked with the Boolean true value, use the string literal `"True"` or `"true"`, the case doesn't matter.

If you're setting up a soft delete policy from the Azure portal, don't add quotes around the soft delete marker value. The field contents are already understood as a string and are translated automatically into a JSON string for you. In the previous examples, simply type `1`, `True` or `true` into the Azure portal's field.

FAQ

Q: Can I index Always Encrypted columns?

No, [Always Encrypted](#) columns aren't currently supported by Azure AI Search indexers.

Q: Can I use Azure SQL indexer with SQL databases running on IaaS VMs in Azure?

Yes. However, you need to allow your search service to connect to your database. For more information, see [Configure a connection from an Azure AI Search indexer to SQL Server on an Azure VM](#).

Q: Can I use Azure SQL indexer with SQL databases running on-premises?

Not directly. We don't recommend or support a direct connection, as doing so would require you to open your databases to Internet traffic. Customers have succeeded with this scenario using bridge technologies like Azure Data Factory. For more information, see [Push data to an Azure AI Search index using Azure Data Factory](#).

Q: Can I use a secondary replica in a [failover cluster](#) as a data source?

It depends. For full indexing of a table or view, you can use a secondary replica.

For incremental indexing, Azure AI Search supports two change detection policies: SQL integrated change tracking and High Water Mark.

On read-only replicas, SQL Database doesn't support integrated change tracking. Therefore, you must use High Water Mark policy.

Our standard recommendation is to use the rowversion data type for the high water mark column. However, using rowversion relies on the `MIN_ACTIVE_ROWVERSION` function, which isn't supported on read-only replicas. Therefore, you must point the indexer to a primary replica if you're using rowversion.

If you attempt to use rowversion on a read-only replica, you get the following error:

"Using a rowversion column for change tracking isn't supported on secondary (read-only) availability replicas. Update the datasource and specify a connection to the primary availability replica. Current database 'Updateability' property is 'READ_ONLY'".

Q: Can I use an alternative, non-rowversion column for high water mark change tracking?

It's not recommended. Only **rowversion** allows for reliable data synchronization. However, depending on your application logic, it can be safe if:

- You can ensure that when the indexer runs, there are no outstanding transactions on the table that's being indexed (for example, all table updates happen as a batch on a

schedule, and the Azure AI Search indexer schedule is set to avoid overlapping with the table update schedule).

- You periodically do a full reindex to pick up any missed rows.
-

Last updated on 11/21/2025

Indexer connections to Azure SQL Managed Instance through a public endpoint

07/11/2025

Indexers in Azure AI Search connect to external data sources over a public endpoint. If you're setting up an [Azure SQL indexer](#) for a connection to a SQL managed instance, follow the steps in this article to ensure the public endpoint is set up correctly.

Alternatively, for private connections, [create a shared private link](#) instead.

 Note

Always Encrypted columns are not currently supported by Azure AI Search indexers.

Enable a public endpoint

This article highlights just the steps for an indexer connection in Azure AI Search. If you want more background, see [Configure public endpoint in Azure SQL Managed Instance](#) instead.

1. For a new SQL Managed Instance, create the resource with the [Enable public endpoint](#) option selected.

Microsoft Azure Search resources, services, and docs (G+) Copilot ...

Home > Create a resource > Marketplace > Azure SQL Managed Instance >

Create Azure SQL Managed Instance

Microsoft

Basics Networking Security Additional settings Tags Review + create

Configure virtual network and public endpoint connectivity for your Managed Instance. Define level of access and connection type. [Learn more](#)

Virtual network

Select or create a virtual network / subnet to connect to your Managed Instance securely. [Learn more](#)

Virtual network / subnet * ⓘ Create new virtual network

Connection type

Select a connection type to accelerate application access. This configuration will apply to virtual network and public endpoint. [Learn more](#)

Connection type (VNet-local endpoint) ⓘ Proxy (Default)

Public endpoint

Secure public endpoint provides the ability to connect to Managed Instance from the Internet without using VPN and is for data communication (TDS) only. Access is disabled by default unless explicitly allowed. [Learn more](#)

Public endpoint (data) ⓘ Disable Enable

Allow access from ⓘ Azure services

Review + create **< Previous** **Next : Security >**

2. Alternatively, if the instance already exists, you can enable public endpoint on an existing SQL Managed Instance under **Security > Networking > Public endpoint > Enable**.

my-sql-mi-0000000 | Networking

SQL managed instance

Search Save Discard Feedback

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Settings

Data management

Security

Networking

Microsoft Defender for Cloud

Subnets selected as a destination for moving the instance that don't contain managed instances are configured as part of the instance move operation. Ensure that any custom rules from the source subnet get configured on the destination subnet as well. Instance move operation will configure only the default values. [Learn more](#)

Public endpoint (data) [i](#)

[Enable](#) [Disable](#)

This option requires port 3342 to be open for inbound traffic. You will need to configure NSG rule for this port separately. [Learn more](#)

Get public endpoint connection string

1. To get a connection string, go to **Settings > Connection strings**.
2. Copy the connection string to use in the search indexer's data source connection. Be sure to copy the connection string for the **public endpoint** (port 3342, not port 1433).

Next steps

With configuration out of the way, you can now specify a SQL managed instance as an indexer data source using the basic instructions for [setting up an Azure SQL indexer](#).

Configure an indexer connection to a SQL Server instance on an Azure virtual machine

05/29/2025

When configuring an [Azure SQL indexer](#) to extract content from a database on an Azure virtual machine, extra steps are required for secure connections.

A connection from Azure AI Search to SQL Server instance on a virtual machine is a public internet connection. In order for secure connections to succeed, perform the following steps:

- Obtain a certificate from a [Certificate Authority provider](#) for the fully qualified domain name of the SQL Server instance on the virtual machine.
- Install the certificate on the virtual machine.

After you install the certificate on your VM, you're ready to complete the following steps in this article.

(!) Note

Always Encrypted columns are not currently supported by Azure AI Search indexers.

Enable encrypted connections

Azure AI Search requires an encrypted channel for all indexer requests over a public internet connection. This section lists the steps to make this work.

1. Check the properties of the certificate to verify the subject name is the fully qualified domain name (FQDN) of the Azure VM.

You can use a tool like CertUtils or the Certificates snap-in to view the properties. You can get the FQDN from the VM service page Essentials section, in the **Public IP address/DNS name label** field, in the [Azure portal](#).

The FQDN is typically formatted as `<your-VM-name>.<region>.cloudapp.azure.com`

2. Configure SQL Server to use the certificate using the Registry Editor (regedit).

Although SQL Server Configuration Manager is often used for this task, you can't use it for this scenario. It won't find the imported certificate because the FQDN of the VM on

Azure doesn't match the FQDN as determined by the VM (it identifies the domain as either the local computer or the network domain to which it's joined). When names don't match, use regedit to specify the certificate.

- a. In regedit, browse to this registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server\  
[MSSQL13.MSSQLSERVER]\MSSQLServer\SuperSocketNetLib\Certificate.
```

The `[MSSQL13.MSSQLSERVER]` part varies based on version and instance name.

- b. Set the value of the **Certificate** key to the **thumbprint** (without spaces) of the TLS/SSL certificate you imported to the VM.

For example, copy the hexadecimal characters to text editor, such as Notepad. Delete all spaces from the thumbprint string. If the thumbprint is `c0 d0 f2 70 95 b0 3d 43 17
e2 19 84 10 24 32 8c ef 24 87 79`, then change it to
`c0d0f27095b03d4317e219841024328cef248779`.

There are several ways to get the thumbprint, some better than others. If you copy it from the **Certificates** snap-in in MMC, you might pick up an invisible leading character, which results in an error when you attempt a connection. Several workarounds exist for correcting this problem. The easiest is to backspace over and then retype the first character of the thumbprint to remove the leading character in the key value field in regedit. Alternatively, you can use a different tool to copy the thumbprint. For more information, see [Certificate thumbprint displayed in MMC certificate snap-in has extra invisible unicode character ↴](#).

3. Grant permissions to the service account.

Make sure the SQL Server service account is granted appropriate permission on the private key of the TLS/SSL certificate. If you overlook this step, SQL Server doesn't start. You can use the **Certificates** snap-in or **CertUtils** for this task.

4. Restart the SQL Server service.

Connect to SQL Server

After you set up the encrypted connection required by Azure AI Search, connect to the instance through its public endpoint. The following article explains the connection requirements and syntax:

- [Connect to SQL Server over the internet](#)

Configure the network security group

It's a best practice to configure the [network security group \(NSG\)](#) and corresponding Azure endpoint or Access Control List (ACL) to make your Azure VM accessible to other parties.

Chances are you've done this before to allow your own application logic to connect to your SQL Azure VM. It's no different for an Azure AI Search connection to your SQL Azure VM.

The following steps and links provide instructions on NSG configuration for VM deployments. Use these instructions to ACL a search service endpoint based on its IP address.

1. Obtain the IP address of your search service. See the [following section](#) for instructions.
2. Add the search IP address to the IP filter list of the security group. Either one of following articles explains the steps:
 - [Tutorial: Filter network traffic with a network security group using the Azure portal](#)
 - [Create, change, or delete a network security group](#)

IP addressing can pose a few challenges that are easily overcome if you're aware of the issue and potential workarounds. The remaining sections provide recommendations for handling issues related to IP addresses in the ACL.

Restrict network access to Azure AI Search

We strongly recommend that you restrict the access to the IP address of your search service and the IP address range of `AzureCognitiveSearch` [service tag](#) in the ACL instead of making your SQL Azure VMs open to all connection requests.

You can find out the IP address by pinging the FQDN (for example, `<your-search-service-name>.search.windows.net`) of your search service. Although it's possible for the search service IP address to change, it's unlikely that it will change. The IP address tends to be static for the lifetime of the service.

You can find out the IP address range of `AzureCognitiveSearch` [service tag](#) by either using [Downloadable JSON files](#) or via the [Service Tag Discovery API](#). The IP address range is updated weekly.

Include the Azure portal IP addresses

If you're using the Azure portal to create an indexer, you must grant the Azure portal inbound access to your SQL Azure virtual machine. An inbound rule in the firewall requires that you provide the IP address of the Azure portal.

To get the Azure portal IP address, ping `stamp2.ext.search.windows.net`, which is the domain of the traffic manager. The request times out, but the IP address is visible in the status message. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is "52.252.175.48".

Clusters in different regions connect to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

Supplement network security with token authentication

Firewalls and network security are a first step in preventing unauthorized access to data and operations. Authorization should be your next step.

We recommend role-based access, where Microsoft Entra ID users and groups are assigned to roles that determine read and write access to your service. See [Connect to Azure AI Search using role-based access controls](#) for a description of built-in roles and instructions for creating custom roles.

If you don't need key-based authentication, we recommend that you disable API keys and use role assignments exclusively.

Next steps

With configuration out of the way, you can now specify a SQL Server on Azure VM as the data source for an Azure AI Search indexer. For more information, see [Index data from Azure SQL](#).

Index data from OneLake files and shortcuts

In this article, learn how to configure a OneLake files indexer for extracting searchable data and metadata data from a [lakehouse](#) on top of [Microsoft OneLake](#).

To configure and run the indexer, you can use:

- [Data Source REST API](#) with an [Indexer REST API](#)
- An Azure SDK package that provides the feature
- [Import data wizard](#) in the Azure portal
- [Import data \(new\) wizard](#) in the Azure portal.

This article uses the REST APIs to illustrate each step.

Prerequisites

- A Fabric workspace. Follow this tutorial to [create a Fabric workspace](#).
- A lakehouse in a Fabric workspace. Follow this tutorial to [create a lakehouse](#).
- Textual data. If you have binary data, you can use [AI enrichment](#) image analysis to extract text or generate descriptions of images. File content can't exceed the [indexer limits](#) for your search service tier.
- Unstructured content in the **Files** location of your lakehouse. You can add data by:
 - [Upload into a lakehouse directly](#)
 - [Use data pipelines from Microsoft Fabric](#)
 - [Add shortcuts](#) from external data sources like [Amazon S3](#) or [Google Cloud Storage](#).
- An AI Search service, basic pricing tier or higher, configured for either a [system managed identity](#) or [user-assigned assigned managed identity](#). The AI Search service must reside within the same tenant as the Microsoft Fabric workspace.
- A Contributor role assignment in the Microsoft Fabric workspace where the lakehouse is located. Steps are outlined in the [Grant permissions](#) section of this article.
- A [REST client](#) to formulate REST calls similar to the ones shown in this article.

Limitations

- Parquet (including delta parquet) file types aren't currently supported.

- File deletion isn't supported for Amazon S3 and Google Cloud Storage shortcuts.
- This indexer doesn't support OneLake workspace Table location content.
- This indexer doesn't support SQL queries, but the query used in the data source configuration is exclusively to add optionally the folder or shortcut to access.
- There's no support to ingest files from **My Workspace** workspace in OneLake since this is a personal repository per user.
- Microsoft Purview sensitivity labels [applied to Fabric items](#) (such as lakehouses) will cause the indexer to fail if the search service doesn't have the required access. To prevent this behavior, you must either:
 - Add the AI Search service's Service Principal Name (SPN) to an existing organization group that grants access under the sensitivity label policy, or
 - Request an exception from your organization's IT team responsible for Purview sensitivity label policy configurations, and have them add the SPN directly to the policy.
- Workspace role-based permissions in Microsoft OneLake may affect indexer access to files. Ensure that the Azure AI Search service principal (managed identity) has sufficient permissions over the files you intend to access in the target [Microsoft Fabric workspace](#).

Supported tasks

You can use this indexer for the following tasks:

- **Data indexing and incremental indexing:** The indexer can index files and associated metadata from data paths within a lakehouse. It detects new and updated files and metadata through built-in change detection. You can configure data refresh on a schedule or on demand.
- **Deletion detection:** The indexer can [detect deletions via custom metadata](#) for most files and shortcuts. This requires adding metadata to files to signify that they have been "soft deleted", enabling their removal from the search index. Currently, it's not possible to detect deletions in Google Cloud Storage or Amazon S3 shortcut files because custom metadata isn't supported for those data sources.
- **Applied AI enrichment through skillsets:** [Skillsets](#) are fully supported by the OneLake files indexer. This includes key features like [integrated vectorization](#) that adds data chunking and embedding steps.
- **Parsing modes:** The indexer supports [JSON parsing modes](#) if you want to parse JSON arrays or lines into individual search documents. It also supports [Markdown parsing mode](#).

- **Compatibility with other features:** The OneLake indexer is designed to work seamlessly with other indexer features, such as [debug sessions](#), [indexer cache for incremental enrichments](#), and [knowledge store](#).

Supported document formats

The OneLake files indexer can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Supported shortcuts

The following OneLake shortcuts are supported by the OneLake files indexer:

- [ADLS Gen2 shortcut](#)
- [OneLake shortcut](#) (a shortcut to another OneLake instance)
- [Amazon S3 shortcut](#)
- [Google Cloud Storage shortcut](#)

Prepare data for indexing

Before you set up indexing, review your source data to determine whether any changes should be made to your data in the lakehouse. An indexer can index content from one container at a

time. By default, all files in the container are processed. You have several options for more selective processing:

- Place files in a virtual folder. An indexer [data source definition](#) includes a "query" parameter that can be either a lakehouse subfolder or shortcut. If this value is specified, only those files in the subfolder or shortcut within the lakehouse are indexed.
- Include or exclude files by file type. The [supported document formats list](#) can help you determine which files to exclude. For example, you might want to exclude image or audio files that don't provide searchable text. This capability is controlled through [configuration settings](#) in the indexer.
- Include or exclude arbitrary files. If you want to skip a specific file for whatever reason, you can add metadata properties and values to files in your lakehouse. When an indexer encounters this property, it skips the file or its content in the indexing run.

File inclusion and exclusion are covered in the [indexer configuration](#) step. If you don't set criteria, the indexer reports an ineligible file as an error and moves on. If enough errors occur, processing might stop. You can specify error tolerance in the indexer [configuration settings](#).

An indexer typically creates one search document per file, where the text content and metadata are captured as searchable fields in an index. If files are whole files, you can potentially parse them into [multiple search documents](#). For example, you can parse rows in a [CSV file](#) to create one search document per row. If you need to chunk a single document into smaller passages to vectorize data, consider using [integrated vectorization](#).

Indexing file metadata

File metadata can also be indexed, and that's helpful if you think any of the standard or custom metadata properties are useful in filters and queries.

User-specified metadata properties are extracted verbatim. To receive the values, you must define field in the search index of type `Edm.String`, with same name as the metadata key of the blob. For example, if a blob has a metadata key of `Priority` with value `High`, you should define a field named `Priority` in your search index and it will be populated with the value `High`.

Standard file metadata properties can be extracted into similarly named and typed fields, as listed below. The OneLake files indexer automatically creates internal field mappings for these metadata properties, converting the original hyphenated name ("metadata-storage-name") to an underscored equivalent name ("metadata_storage_name").

You still have to add the underscored fields to the index definition, but you can omit [indexer field mappings](#) because the indexer makes the association automatically.

- **metadata_storage_name** (`Edm.String`) - the file name. For example, if you have a file `/mydatalake/my-folder/subfolder/resume.pdf`, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the blob, including the storage account. For example, `https://myaccount.blob.core.windows.net/my-container/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the blob. For example, `application/octet-stream`.
- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the blob. Azure AI Search uses this timestamp to identify changed blobs, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - blob size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the blob content, if available.

Lastly, any metadata properties specific to the document format of the files you're indexing can also be represented in the index schema. For more information about content-specific metadata, see [Content metadata properties](#).

It's important to point out that you don't need to define fields for all of the above properties in your search index - just capture the properties you need for your application.

Grant permissions

The OneLake indexer uses token authentication and role-based access for connections to OneLake. Permissions are assigned in OneLake. There are no permission requirements on the physical data stores backing the shortcuts. For example, if you're indexing from AWS, you don't need to grant search service permissions in AWS.

The minimum role assignment for your search service identity is Contributor.

1. [Configure a system or user-managed identity](#) for your AI Search service.

The following screenshot shows a system managed identity for a search service named "onelake-demo".

This screenshot shows the Identity settings for a search service named 'onelake-demo'. The 'System assigned' tab is selected. A red box highlights the 'Status' switch, which is set to 'On'. Another red box highlights the 'User assigned' link in the top navigation bar.

This screenshot shows a user-managed identity for the same search service.

This screenshot shows the Identity settings for the same search service 'onelake-demo'. The 'User assigned' tab is selected. A red box highlights the 'demo-mi' entry in the list of identities.

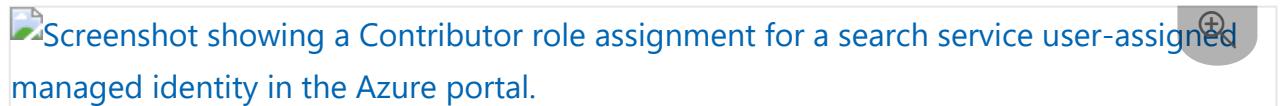
2. Grant permission for search service access to the Fabric workspace. The search service makes the connection on behalf of the indexer.

If you use a system-assigned managed identity, search for the name of the AI Search service. For a user-assigned managed identity, search for the name of the identity resource.

The following screenshot shows a Contributor role assignment using a system managed identity.



This screenshot shows a Contributor role assignment using a user-assigned managed identity:



Configure a shared private link (required if using Fabric workspace-level private link)

If your Fabric workspace is secured with a [private link](#), Azure AI Search won't be able to access your lakehouse data over the public internet, and you won't be able to configure the indexer or its required dependencies, such as the data source. To enable access, you must configure a [shared private link](#) between Azure AI Search and your Fabric workspace.

Define the data source

A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Use the [Create or update a data source REST API](#) to set its definition. These are the most significant steps of the definition.
 2. Set "type" to "onelake" (required).
 3. Get the Microsoft Fabric workspace GUID and the lakehouse GUID:
 - In Power BI, open the lakehouse you'd like to import data from. Notice the lakehouse URL in the browser. It should look similar to this example: "<https://msit.powerbi.com/groups/00000000-0000-0000-0000-000000000000/lakehouses/11111111-1111-1111-1111-111111111111>". The URL contains both the workspace GUID and the lakehouse GUID. If the Fabric workspace is secured with a private link, the URL would start with "<https://{{FabricWorkspaceGuid}}.z{{xy}}.blob.fabric.microsoft.com>".
 - Copy the workspace GUID, which is listed to the right of "groups" in the URL. In this example, it would be 00000000-0000-0000-000000000000. In your REST file, create an environment variable for {{FabricWorkspaceGuid}} and set it to the workspace GUID. If your workspace uses a private link, the workspace GUID will appear in a different location in the URL. Be sure to reference the correct part of the URL based on your setup.

A screenshot of the Microsoft Power BI Groups interface. The URL in the browser's address bar is `https://msit.powerbi.com/groups/00000000-0000-0000-0000-000000000000/lakehouses/11111111-1111-1111-1111-111111111111`. A red arrow points from the text "Copy the lakehouse GUID" in the instructions below to the GUID in the URL.

- Copy the lakehouse GUID, which is listed right after "lakehouses" in the URL. In this example, it would be `11111111-1111-1111-1111-111111111111`. In your REST file, create an environment variable for `{LakehouseGuid}` and set it to the lakehouse GUID.

A screenshot of the Microsoft Power BI Groups interface, similar to the one above but with a different URL. The URL in the browser's address bar is `https://msit.powerbi.com/groups/00000000-0000-0000-0000-000000000000/lakehouses/11111111-1111-1111-1111-111111111111`. A red arrow points from the text "Set 'credentials'" in the instructions below to the GUID in the URL.

- Set "credentials" to the Microsoft Fabric workspace GUID by replacing `{FabricWorkspaceGuid}` with the value you copied in the previous step. This is the OneLake to access with the managed identity you'll set up later in this guide.

JSON

```
"credentials": {  
  "connectionString": "ResourceId={FabricWorkspaceGuid}"  
}
```

For your setup with [shared private link](#), setup the managed identities using the following connection string, that varies from the setup using the internet for communication. Note that not only the URL is different, but also `WorkspaceEndpoint` is used, instead of `ResourceId`. Take this into consideration when configuring either the system-managed identity or user-managed identity setups.

JSON

```
"credentials": {  
  "connectionString":  
    "WorkspaceEndpoint=https://{{FabricWorkspaceGuid}}.z{{xy}}.blob.fabric.microsoft.com"  
}
```

1. Set `"container.name"` to the lakehouse GUID, replacing `{LakehouseGuid}` with the value you copied in the previous step. Use `"query"` to optionally specify a lakehouse subfolder or shortcut.

JSON

```
"container": {  
  "name": "{LakehouseGuid}",  
  "query": "{optionalLakehouseFolderOrShortcut}"  
}
```

2. Set the authentication method using the user-assigned managed identity, or skip to the next step for system-managed identity.

JSON

```
{  
  "name": "{dataSourceName}",  
  "description": "description",  
  "type": "onelake",  
  "credentials": {  
    "connectionString": "ResourceId={{FabricWorkspaceGuid}}"  
  },  
  "container": {  
    "name": "{LakehouseGuid}",  
    "query": "{optionalLakehouseFolderOrShortcut}"  
  },  
  "identity": {  
    "@odata.type": "Microsoft.Azure.Search.DataUserAssignedIdentity",  
    "userAssignedIdentity": "{userAssignedManagedIdentity}"  
  }  
}
```

The `userAssignedIdentity` value can be found by accessing the `{userAssignedManagedIdentity}` resource, under Properties and it's called `Id`.



contoso-user-managed-id | Properties



Managed Identity

Search



Refresh

Overview

Activity log

Access control (IAM)

Tags

Azure role assignments

Associated resources (preview)

Settings

Federated credentials

Properties

Locks

> Monitoring



Essentials

Id	/subscriptions/ aaaa0a0a-bb1b-...
Name	contoso-user-managed-id
Type	Microsoft.ManagedIdentity/user...
Tags	View value as JSON
Location	eastus



Example:

JSON

```
{  
  "name": "mydatasource",  
  "description": "description",  
  "type": "onelake",  
  "credentials": {  
    "connectionString": "ResourceId=a0a0a0a0-bbbb-cccc-dddd-e1e1e1e1e1"  
  },  
  "container": {  
    "name": "11111111-1111-1111-1111-111111111111",  
    "query": "folder_name"  
  },  
  "identity": {  
    "@odata.type": "Microsoft.Azure.Search.DataUserAssignedIdentity",  
    "userAssignedIdentity": "/subscriptions/333333-3333-3333-  
33333333/resourcegroups/myresourcegroup/providers/Microsoft.ManagedIdentity/use  
rAssignedIdentities/demo-mi"  
  }  
}
```

3. Optionally, use a system-assigned managed identity instead. The "identity" is removed from the definition if using system-assigned managed identity.

JSON

```
{
  "name": "{dataSourceName}",
  "description": "description",
  "type": "onelake",
  "credentials": {
    "connectionString": "ResourceId={FabricWorkspaceGuid}"
  },
  "container": {
    "name": "{LakehouseGuid}",
    "query": "{optionalLakehouseFolderOrShortcut}"
  }
}
```

Example:

JSON

```
{
  "name": "mydatasource",
  "description": "description",
  "type": "onelake",
  "credentials": {
    "connectionString": "ResourceId=a0a0a0a0-bbbb-cccc-dddd-e1e1e1e1e1"
  },
  "container": {
    "name": "1111111-1111-1111-1111-111111111111",
    "query": "folder_name"
  }
}
```

Detect deletions via custom metadata

The OneLake files indexer data source definition can include a [soft deletion policy](#) if you want the indexer to delete a search document when the source document is flagged for deletion.

To enable automatic file deletion, use custom metadata to indicate whether a search document should be removed from the index.

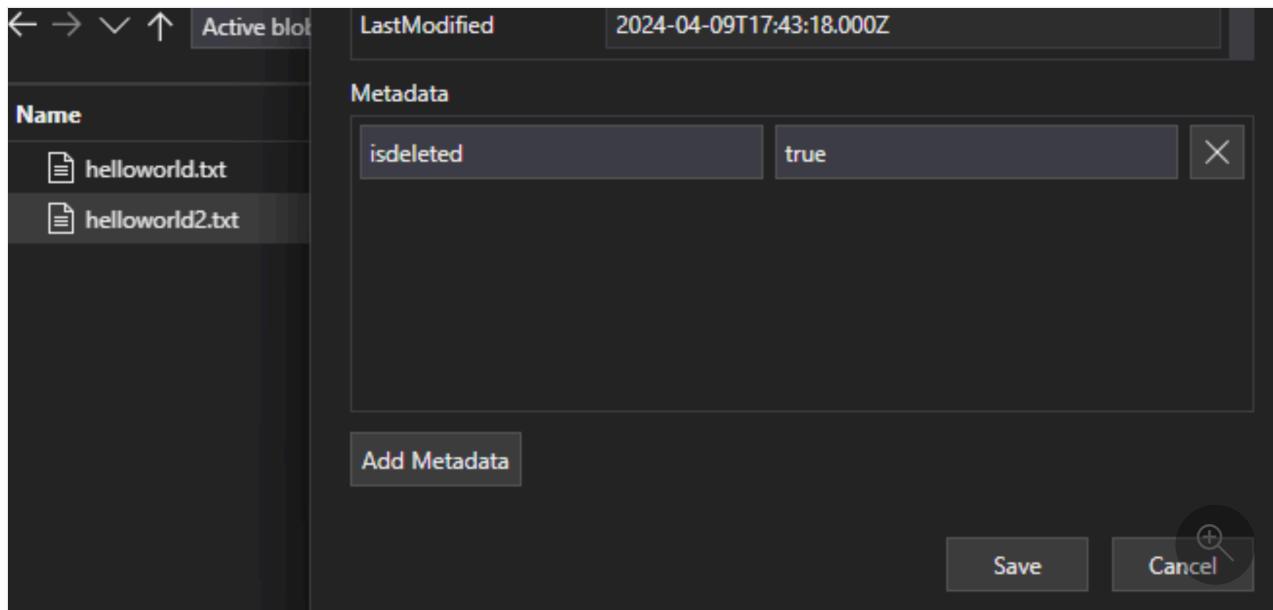
Workflow requires three separate actions:

- "Soft-delete" the file in OneLake
- Indexer deletes the search document in the index
- "Hard delete" the file in OneLake

"Soft-deleting" tells the indexer what to do (delete the search document). If you delete the physical file in OneLake first, there's nothing for the indexer to read and the corresponding search document in the index is orphaned.

There are steps to follow in both OneLake and Azure AI Search, but there are no other feature dependencies.

1. In the lakehouse file, add a custom metadata key-value pair to the file to indicate the file is flagged for deletion. For example, you could name the property "IsDeleted", set to false. When you want to delete the file, change it to true.



2. In Azure AI Search, edit the data source definition to include a "dataDeletionDetectionPolicy" property. For example, the following policy considers a file to be deleted if it has a metadata property "IsDeleted" with the value true:

https

```
PUT https://[service name].search.windows.net/datasources/file-datasource?api-version=2025-09-01
{
    "name" : "onelake-datasource",
    "type" : "onelake",
    "credentials": {
        "connectionString": "ResourceId={FabricWorkspaceGuid}"
    },
    "container": {
        "name": "{LakehouseGuid}",
        "query": "{optionalLakehouseFolderOrShortcut}"
    },
    "dataDeletionDetectionPolicy" : {
        "@odata.type"
        :"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName" : "IsDeleted",
        "softDeleteMarkerValue" : "true"
    }
}
```

After the indexer runs and deletes the document from the search index, you can then delete the physical file in the data lake.

Some key points include:

- [Scheduling an indexer run](#) helps automate this process. We recommend schedules for all incremental indexing scenarios.
- If the deletion detection policy wasn't set on the first indexer run, you must [reset the indexer](#) so that it reads the updated configuration.
- Recall that deletion detection isn't supported for Amazon S3 and Google Cloud Storage shortcuts due to the dependency on custom metadata.

Add search fields to an index

In a [search index](#), add fields to accept the content and metadata of your OneLake data lake files.

1. [Create or update an index](#) to define search fields that store file content and metadata:

HTTP

```
{  
    "name" : "my-search-index",  
    "fields": [  
        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false  
    },  
        { "name": "content", "type": "Edm.String", "searchable": true,  
        "filterable": false },  
        { "name": "metadata_storage_name", "type": "Edm.String", "searchable":  
        false, "filterable": true, "sortable": true },  
        { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable":  
        false, "filterable": true, "sortable": true },  
        { "name": "metadata_storage_content_type", "type": "Edm.String",  
        "searchable": false, "filterable": true, "sortable": true }  
    ]  
}
```

2. Create a document key field ("key": true). For file content, the best candidates are metadata properties.

- `metadata_storage_path` (default) full path to the object or file. The key field ("ID" in this example) is populated with values from `metadata_storage_path` because it's the default.

- `metadata_storage_name`, usable only if names are unique. If you want this field as the key, move `"key": true` to this field definition.
- A custom metadata property that you add to your files. This option requires that your file upload process adds that metadata property to all blobs. Since the key is a required property, any files that are missing a value fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers add duplicate documents for the same file if the key property changes.

Metadata properties often include characters, such as `/` and `-`, that are invalid for document keys. Because the indexer has a `"base64EncodeKeys"` property (true by default), it automatically encodes the metadata property, with no configuration or field mapping required.

3. Add a "content" field to store extracted text from each file through the file's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.
4. Add fields for standard metadata properties. The indexer can read custom metadata properties, [standard metadata](#) properties, and [content-specific metadata](#) properties.

Configure and run the OneLake files indexer

Once the index and data source are created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors. You can also specify which parts of a blob to index.

1. [Create or update an indexer](#) by giving it a name and referencing the data source and target index:

JSON

```
{
  "name" : "my-onelake-indexer",
  "dataSourceName" : "my-onelake-datasource",
  "targetIndexName" : "my-search-index",
  "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null,
    "base64EncodeKeys": null,
    "configuration": {
      "indexedFileNameExtensions" : ".pdf,.docx",
      "excludedFileNameExtensions" : ".png,.jpeg",
      "dataToExtract": "contentAndMetadata",
      "parsingMode": "default"
    }
  }
}
```

```
        }
    },
    "schedule" : { },
    "fieldMappings" : [ ]
}
```

2. Set "batchSize" if the default (10 documents) is either under utilizing or overwhelming available resources. Default batch sizes are data source specific. File indexing sets batch size at 10 documents in recognition of the larger average document size.
3. Under "configuration", control which files are indexed based on file type, or leave unspecified to retrieve all files.

For "indexedFileNameExtensions", provide a comma-separated list of file extensions (with a leading dot). Do the same for "excludedFileNameExtensions" to indicate which extensions should be skipped. If the same extension is in both lists, it's excluded from indexing.

4. Under "configuration", set "dataToExtract" to control which parts of the files are indexed:
 - "contentAndMetadata" is the default. It specifies that all metadata and textual content extracted from the file are indexed.
 - "storageMetadata" specifies that only the [standard file properties and user-specified metadata](#) are indexed. Although the properties are documented for Azure blobs, the file properties are the same for OneLake, except for the SAS related metadata.
 - "allMetadata" specifies that standard file properties and any [metadata for found content types](#) are extracted from the file content and indexed.
5. Under "configuration", set "parsingMode" if files should be mapped to [multiple search documents](#), or if they consist of [plain text](#), [JSON documents](#), or [CSV files](#).
6. [Specify field mappings](#) if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

In file indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer automatically replaces hyphens - with underscores in the search index.

For more information about other properties, [Create an indexer](#). For the full list of parameter descriptions, see [Create Indexer \(REST\)](#) in the REST API. The parameters are the same for Microsoft OneLake.

By default, an indexer runs automatically when you create it. You can change this behavior by setting "disabled" to true. If you create an indexer in a disabled state, [run an indexer on demand](#) when you're ready to use it, or [put it on a schedule](#).

Check indexer status

Learn multiple approaches to [monitor the indexer status and execution history here](#).

Handle errors

Errors that commonly occur during indexing include unsupported content types, missing content, or oversized files. By default, the OneLake files indexer stops as soon as it encounters a file with an unsupported content type. However, you might want indexing to proceed even if errors occur, and then debug individual documents later.

Transient errors are common for solutions involving multiple platforms and products. However, if you keep the [indexer on a schedule](#) (for example every 5 minutes), the indexer should be able to recover from those errors in the following run.

There are five indexer properties that control the indexer's response when errors occur.

JSON

```
{  
  "parameters" : {  
    "maxFailedItems" : 10,  
    "maxFailedItemsPerBatch" : 10,  
    "configuration" : {  
      "failOnUnsupportedContentType" : false,  
      "failOnUnprocessableDocument" : false,  
      "indexStorageMetadataOnlyForOversizedDocuments": false  
    }  
  }  
}
```

[] [Expand table](#)

Parameter	Valid values	Description
"maxFailedItems"	-1, null or 0, positive integer	Continue indexing if errors happen at any point of processing, either while parsing blobs or while adding documents to an index. Set these properties to the number of

Parameter	Valid values	Description
		acceptable failures. A value of <code>-1</code> allows processing no matter how many errors occur. Otherwise, the value is a positive integer.
<code>"maxFailedItemsPerBatch"</code>	<code>-1</code> , null or 0, positive integer	Same as above, but used for batch indexing.
<code>"failOnUnsupportedContentType"</code>	true or false	If the indexer is unable to determine the content type, specify whether to continue or fail the job.
<code>"failOnUnprocessableDocument"</code>	true or false	If the indexer is unable to process a document of an otherwise supported content type, specify whether to continue or fail the job.
<code>"indexStorageMetadataOnlyForOversizedDocuments"</code>	true or false	Oversized blobs are treated as errors by default. If you set this parameter to true, the indexer tries to index its metadata even if the content can't be indexed. For limits on blob size, see service Limits .

Next steps

Review how the [Import data \(new\) wizard](#) works and try it out for this indexer. You can use [integrated vectorization](#) to chunk and create embeddings for vector or hybrid search using a default schema.

Last updated on 11/21/2025

Index data from SharePoint document libraries

ⓘ Important

SharePoint in Microsoft 365 indexer support is in public preview. It's offered "as-is", under [Supplemental Terms of Use](#) and supported on best effort only. Preview features aren't recommended for production workloads and aren't guaranteed to become generally available.

See [known limitations](#) section before you start.

[Fill out this form](#) to register for the preview. All requests are approved automatically. After you fill out the form, use a [preview REST API](#) to index your content.

This article explains how to configure a [search indexer](#) to index documents stored in SharePoint document libraries for full text search in Azure AI Search. Configuration steps are first, followed by behaviors and scenarios.

In Azure AI Search, an indexer extracts searchable data and metadata from a data source. The SharePoint in Microsoft 365 indexer provides the following functionality:

- Indexes files and metadata from one or more document libraries.
- Indexes incrementally, picking up just the new and changed files and metadata.
- Detects deleted content automatically. Document deletion in the library is picked up on the next indexer run, and the corresponding search document is removed from the index.
- Extracts text and normalized images from indexed documents automatically. Optionally, you can add a [skillset](#) for deeper [AI enrichment](#), such as optical character recognition (OCR) or entity recognition.
- Supports document [basic Access Control Lists \(ACL\) ingestion](#) in public preview during initial document sync. It also supports full data set incremental data sync.
- Supports [Microsoft Purview sensitivity label ingestion and honoring at query time](#). This functionality is in public preview.

Prerequisites

- [Azure AI Search](#), Basic pricing tier or higher.
- [SharePoint in Microsoft 365](#) cloud service (OneDrive isn't a supported data source).
- Files in a [document library](#).
- [Visual Studio Code](#) with the [REST Client extension](#) for setting up and running the indexer pipeline.

Supported document formats

The SharePoint in Microsoft 365 indexer can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Limitations and considerations

Here are the limitations of this feature:

- The indexer can index content from supported document formats in a document library. There's no indexer support for [SharePoint Lists](#), .ASX site content, or OneNote notebook files. Furthermore, indexing sub-sites recursively from a specific site isn't supported.
- Incremental indexing limitations:
 - Renaming a SharePoint folder breaks incremental indexing. A renamed folder is treated as new content.
 - Microsoft 365 processes that update SharePoint file system metadata can trigger incremental indexing, even if there are no other changes to content. Test your setup and check your document processing behaviors in Microsoft 365 platform before using the indexer or any AI enrichment.
- Security limitations:
 - No support for [Private endpoint](#). Secure network configuration must be enabled [via a firewall](#).
 - No support for tenants with [Microsoft Entra ID Conditional Access](#) enabled.
 - User-encrypted files and password-protected ZIP files aren't supported. However, encrypted content is allowed if it's protected by [Purview sensitivity labels](#) and if the [configuration to preserve and honor those labels \(preview\)](#) is enabled.
 - Limited support for document-level access permissions. A basic level of Access Control Lists (ACL) sync is currently in public preview. Review the [SharePoint ACL configuration documentation](#) for details and setup.

① Important

Don't test or enable both [SharePoint ACL ingestion \(preview\)](#) and [preserving and honoring sensitivity labels \(preview\)](#) in the same indexer or index during preview. Use separate indexers / indexes for each feature. Coexistence of both features is not currently supported at this time.

Here are some considerations when using this feature:

- To build a custom Copilot or Retrieval-Augmented Generation (RAG) app that interacts with SharePoint data using Azure AI Search, Microsoft recommends using the [SharePoint \(Remote\) Knowledge Source](#). This knowledge source uses the [Copilot Retrieval API](#) to query textual content directly from SharePoint in Microsoft 365, returning results to the agentic retrieval engine for merging, ranking, and response formulation. There's no search index used by this knowledge source, and only textual content is queried. Azure AI Search doesn't replicate data. It enforces the SharePoint permission model by returning only the results that each user is authorized to see.
- If you need to create a custom Copilot / RAG (Retrieval Augmented Generation) application or AI agent to chat with SharePoint data in production environments, consider first building it directly via [Microsoft Copilot Studio](#).
- If you still need a custom copilot / RAG application or agent indexing data from SharePoint in Azure AI Search in a production environment, despite the recommendation to use Copilot Studio, consider:
 - Creating a custom connector with [SharePoint Webhooks](#), calling [Microsoft Graph API](#) to export the data to an Azure Blob container, and then use the [Azure blob indexer](#) for incremental indexing.
 - Creating your own [Azure Logic Apps workflow](#) using [Azure Logic Apps SharePoint connector](#) and [Azure AI Search connector](#) when reaching General Availability. You can use the workflow generated by the [Azure portal wizard](#) as a starting point and then customize it in the [Azure Logic Apps designer](#) to include the transformation steps you need. The Azure Logic App workflow created when using the [Azure AI Search wizard](#) to index SharePoint in Microsoft 365 data is a [consumption workflow](#). When setting up production workloads, switch to a [standard logic app workflow](#) to use its extra enterprise features.

Regardless of the approach you choose, whether building a custom connector with SharePoint hooks or creating an Azure Logic Apps workflow, be sure to implement robust security measures. These measures include configuring shared private links, setting up firewalls, preserving user permissions from the source and honor those permissions at query time, among others. You should also regularly audit and monitor your pipeline.

Configure the SharePoint in Microsoft 365 indexer

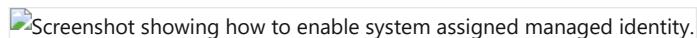
To set up the SharePoint in Microsoft 365 indexer, use a preview REST API. This section provides the steps.

Step 1 (Optional): Enable system assigned managed identity

Enable a [system-assigned managed identity](#) to automatically detect the tenant in which the search service is provisioned.

Perform this step if the SharePoint site is in the same tenant as the search service. Skip this step if the SharePoint site is in a different tenant. The identity is used for tenant detection. You can also skip this step if you want to put the tenant ID in the [connection string](#). If you would like to use

the system-managed identity or configure a user-assigned managed identity for secretless indexing, configure the [application permissions with secretless authentication](#)



After selecting **Save**, you receive an Object ID assigned to your search service.

Step 2: Decide which permissions the indexer requires

The SharePoint in Microsoft 365 indexer supports both [delegated and application](#) permissions. Choose which permissions you want to use based on your scenario.

We recommend app-based permissions. See [limitations](#) for known issues related to delegated permissions.

- Application permissions (recommended), where the indexer runs under the [identity of the SharePoint tenant](#) with access to all sites and files. The indexer requires a [client secret](#). The indexer also requires [tenant admin approval](#) before it can index content. This permission type is the only one that supports basic [ACL preservation \(preview\)](#) configuration. Delegated permissions can't be used for ACL sync.
- Delegated permissions, where the indexer runs under the identity of the user or app sending the request. Data access is limited to the sites and files to which the caller has access. To support delegated permissions, the indexer requires a [device code prompt](#) to sign in on behalf of the user. User-delegated permissions enforce token expiration every 75 minutes, per the most recent security libraries used to implement this authentication type. This isn't a behavior that can be adjusted. An expired token requires manual indexing using [Run Indexer \(preview\)](#). For this reason, you should use app-based permissions instead. This configuration is only recommended for small testing operations, due to token expiration period and since this permission type doesn't support any level of [ACL preservation](#) configuration.

Step 3: Create a Microsoft Entra application registration

The SharePoint in Microsoft 365 indexer uses a Microsoft Entra application for authentication. Create the application registration in the same tenant as Azure AI Search.

- Sign in to the [Azure portal](#).
- Search for or navigate to **Microsoft Entra ID**, then select **Add > App registrations**.
- Select + New registration:
 - Provide a name for your app.
 - Select **Single tenant**.
 - Skip the URI designation step. No redirect URI required.
 - Select **Register**.
- On the navigation pane under **Manage**, select **API permissions**, then **Add a permission**, then **Microsoft Graph**.
 - If your indexer uses application API permissions, choose **Application** permissions.
 - For standard indexing, select: `Files.Read.All` `Sites.Read.All`
 - If you're enabling content indexing and [basic ACL sync \(preview\)](#), select: `Files.Read.All` `Sites.FullControl.All` (instead of `Sites.Read.All`)
 - If you need to enable content indexing and limit [ACL sync \(preview\)](#) to specific sites, select: `Files.Read.All` `Sites.Selected` Then grant the application full control only for those selected sites.



Using application permissions means that the indexer accesses the SharePoint site in a service context. So when you run the indexer, it has access to all content in the SharePoint tenant, which requires tenant admin approval. A client secret or secretless configuration is also required for authentication. Setting up the authentication mechanism is described later in this article under [authentication modes for application API permissions only](#).

- If the indexer is using delegated API permissions, select **Delegated permissions** and then select `Delegated - Files.Read.All`, `Delegated - Sites.Read.All`, and `Delegated - User.Read`.

Add a permission				
API / Permissions name	Type	Description	Admin consent req...	Status
Microsoft Graph (3)				
Files.Read.All	Delegated	Read all files that user can access	-	...
Sites.Read.All	Delegated	Read items in all site collections	-	...
User.Read	Delegated	Sign in and read user profile	-	...

Delegated permissions allow the search client to connect to SharePoint under the security identity of the current user.

5. Give admin consent.

Tenant admin consent is required when using application API permissions. Some tenants are locked down in such a way that tenant admin consent is required for delegated API permissions as well. If either of these conditions apply, you'll need to have a tenant admin grant consent for this Microsoft Entra application before creating the indexer.



6. Select the Authentication tab.

7. Set Allow public client flows to Yes then select Save.

8. Select + Add a platform, then Mobile and desktop applications, then check

<https://login.microsoftonline.com/common/oauth2/nativeclient>, then **Configure**.



9. Configure the indexer **authentication method** according to your solution needs.

Available authentication methods for application API permissions only

To authenticate the Microsoft Entra application with application permissions, the indexer uses either a client secret or a secretless configuration.

Using client secret

These are the instructions to configure the application to use a client secret to authenticate the indexer, so it can ingest data from SharePoint.

- Select Certificates & Secrets from the menu on the left, then Client secrets, then New client secret.



- In the menu that pops up, enter a description for the new client secret. Adjust the expiration date if necessary. If the secret expires, it needs to be recreated and the indexer needs to be updated with the new secret.

Add a client secret

Description	SharePoint Online Indexer
Expires	Recommended: 6 months
<input type="button" value="Add"/> <input type="button" value="Cancel"/>	

- The new client secret appears in the secret list. Once you navigate away from the page, the secret is no longer be visible, so copy the value using the copy button and save it in a secure location.



Using secretless authentication to obtain application tokens

These are the instructions to configure the application so Microsoft Entra trusts a managed identity to obtain an application token to authenticate without a client secret, so the indexer can ingest data from SharePoint.

Configuring the registered application with a managed identity

1. Create (or select) a [user-assigned managed identity](#) and assign to your search service or a [system-assigned managed identity](#), depending on your scenario requirements.
2. Capture the **object (principal)** ID. This will be used as part of the credentials configuration when creating the data source.
3. Select **Certificates & Secrets** from the menu on the left.
4. Under **Federated credentials** select + **Add a credential**.
5. Under **Federated credential scenario** select **Managed Identity**.
6. Select managed identity: Choose the created managed identity created as part of step 1.
7. Add a name for your credential and click on **Save**.

Step 4: Create data source

Starting in this section, use the latest preview REST API and a REST client or the latest supported beta SDK of your preference for the remaining steps.

A data source specifies which data to index, credentials, and policies to efficiently identify changes in the data (new, modified, or deleted rows). Multiple indexers in the same search service can use the same data source.

For SharePoint indexing, the data source must have the following required properties:

- **name** is the unique name of the data source within your search service.
- **type** must be "sharepoint". This value is case-sensitive.
- **credentials** provide the SharePoint endpoint and the authentication method allowed for the application to request the Microsoft Entra tokens. An example SharePoint endpoint is <https://microsoft.sharepoint.com/teams/MySharePointSite>. You can get the endpoint by navigating to the home page of your SharePoint site and copying the URL from the browser. Review the [connection string format](#) for the supported syntax.
- **container** specifies which document library to index. Properties [control which documents are indexed](#).

To create a data source, call [Create Data Source \(preview\)](#).

Here's a data source definition sample for credentials with application secret or service-assigned managed identity.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-datasource",
    "type" : "sharepoint",
    "credentials" : { "connectionString" : "[connection-string]" },
    "container" : { "name" : "defaultSiteLibrary", "query" : null }
}
```

Here's a data source definition sample for credentials with user-assigned managed identity.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-datasource",
    "type" : "sharepoint",
    "credentials" : { "connectionString" : "[connection-string]" },
    "container" : { "name" : "defaultSiteLibrary", "query" : null },
    "identity": {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity": "/subscriptions/[Azure Subscription ID]/resourceGroups/[resource-group]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[user-assigned managed identity]"
    }
}
```

Connection string format

The format of the connection string changes based on whether the indexer is using delegated API permissions or application API permissions.

- Delegated API permissions connection string format

```
SharePointOnlineEndpoint=[SharePoint site url];ApplicationId=[Azure AD App ID];TenantId=[SharePoint site tenant id]
```

- Application API permissions with application secret connection string format

```
SharePointOnlineEndpoint=[SharePoint site url];ApplicationId=[Azure AD App ID];ApplicationSecret=[Azure AD App client secret];TenantId=[SharePoint site tenant id]
```

- Application API permissions with secretless (system-assigned managed identity) connection string format

```
SharePointOnlineEndpoint=[SharePoint site url];ApplicationId=[Azure AD App ID];FederatedCredentialObjectId=[selected managed identity object (principal) ID];TenantId=[SharePoint site tenant id]
```

You can get `TenantId` from the overview page in the Microsoft Entra admin center in your Microsoft 365 subscription. You can get the managed identity `object (principal) ID` from the section [Configuring the registered application with a managed identity](#)

① Note

If the SharePoint site is in the same tenant as the search service and system-assigned managed identity is enabled, `TenantId` doesn't have to be included in the connection string. If the SharePoint site is in a different tenant from the search service, `TenantId` must be included.

If your indexer uses [SharePoint ACL configuration \(preview\)](#) or [preserves and honors Microsoft Purview sensitivity labels \(preview\)](#), review the related articles for data source setup before creating the indexer. Each feature has specific configuration steps.

Step 5: Create an index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

To create an index, call [Create Index \(preview\)](#):

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-index",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "metadata_spo_item_name", "type": "Edm.String", "key": false, "searchable": true, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "metadata_spo_item_path", "type": "Edm.String", "key": false, "searchable": false, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "metadata_spo_item_content_type", "type": "Edm.String", "key": false, "searchable": false, "filterable": true,
"sortable": false, "facetable": true },
        { "name": "metadata_spo_item_last_modified", "type": "Edm.DateTimeOffset", "key": false, "searchable": false, "filterable": false,
"sortable": true, "facetable": false },
        { "name": "metadata_spo_item_size", "type": "Edm.Int64", "key": false, "searchable": false, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false }
    ]
}
```

① Important

Only [metadata spo site library item id](#) may be used as the key field in an index populated by the SharePoint in Microsoft 365 indexer. If a key field doesn't exist in the data source, `metadata_spo_site_library_item_id` is automatically mapped to the key field.

If your indexer will use either [SharePoint ACL configuration \(preview\)](#) or will [preserve and honor Microsoft Purview sensitivity labels \(preview\)](#), review each article for index and skillset configuration before proceeding with indexer creation since those functionalities have specific configurations.

Step 6: Create an indexer

An indexer connects a data source with a target search index and provides a schedule to automate the data refresh. Once the index and data source are created, you can create the indexer.

If you're using delegated permissions, during this step, you're asked to sign in with organization credentials that have access to the SharePoint site. If possible, we recommend creating a new organizational user account and giving that new user the exact permissions that you want the indexer to have.

There are a few steps to creating the indexer:

1. Send a [Create Indexer \(preview\)](#) request:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-indexer",
    "dataSourceName" : "sharepoint-datasource",
    "targetIndexName" : "sharepoint-index",
    "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "base64EncodeKeys": null,
        "maxFailedItemsPerBatch": null,
        "configuration": {
            "indexedFileNameExtensions" : ".pdf, .docx",
            "excludedFileNameExtensions" : ".png, .jpg",
            "dataToExtract": "contentAndMetadata"
        }
    },
    "schedule" : { },
    "fieldMappings" : [
        {
            "sourceFieldName" : "metadata_spo_site_library_item_id",
            "targetFieldName" : "id",
            "mappingFunction" : {
                "name" : "base64Encode"
            }
        }
    ]
}
```

If you're using application permissions, it's necessary to wait until the initial run is complete before starting to query your index. The following instructions provided in this step pertain specifically to delegated permissions, and aren't applicable to application permissions.

2. When you create the indexer for the first time, the [Create Indexer \(preview\)](#) request waits until you complete the next step. You must call [Get Indexer Status](#) to get the link and enter your new device code.

HTTP

```
GET https://[service name].search.windows.net/indexers/sharepoint-indexer/status?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]
```

If you don't run the [Get Indexer Status](#) within 10 minutes, the code expires and you'll need to recreate the [data source](#).

3. Copy the device login code from the [Get Indexer Status](#) response. The device login can be found in the "errorMessage".

HTTP

```
{
    "lastResult": {
        "status": "transientFailure",
        "errorMessage": "To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code <CODE> to authenticate."
    }
}
```

4. Provide the code that was included in the error message.



5. The SharePoint in Microsoft 365 indexer will access the SharePoint content as the signed-in user. The user that logs in during this step will be that signed-in user. So, if you sign in with a user account that doesn't have access to a document in the Document Library that you want to index, the indexer won't have access to that document.

If possible, we recommend creating a new user account and giving that new user the exact permissions that you want the indexer to have.

6. Approve the permissions that are being requested.



7. The [Create Indexer \(preview\)](#) initial request completes if all the permissions provided above are correct and within the 10 minute timeframe.

Note

If the Microsoft Entra application requires admin approval and was not approved before logging in, you may see the following screen. [Admin approval](#) is required to continue. A screenshot of a web browser showing a consent screen from Microsoft Entra. It asks for permission to access the user's profile information. The user has selected 'Allow' and is prompted to 'Sign in with Microsoft'. Below the consent screen, there is a message: 'Admin approval is required to continue. Please sign in with your organization's administrator account.' There is also a link to 'Get help'.

Step 7: Check the indexer status

After the indexer has been created, you can call [Get Indexer Status](#):

HTTP

```
GET https://[service name].search.windows.net/indexers/sharepoint-indexer/status?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]
```

Updating the data source

If there are no updates to the data source object, the indexer runs on a schedule without any user interaction.

If you change the data source while the device code is expired, sign in again to run the indexer. For example, if you change the data source query, sign in again using the <https://microsoft.com/devicelogin> and get the new device code.

Here are the steps for updating a data source, assuming an expired device code:

1. Call [Run Indexer \(preview\)](#) to manually start [indexer execution](#).

HTTP

```
POST https://[service name].search.windows.net/indexers/sharepoint-indexer/run?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]
```

2. Check the [indexer status](#).

HTTP

```
GET https://[service name].search.windows.net/indexers/sharepoint-indexer/status?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]
```

3. If you get an error asking you to visit <https://microsoft.com/devicelogin>, open the page and copy the new code.

4. Paste the code into the dialog box.

5. Manually run the indexer again and check the indexer status. This time, the indexer run should successfully start.

Indexing document metadata

If you're indexing document metadata (`"dataToExtract": "contentAndMetadata"`), the following metadata is available to index.

Expand table

Identifier	Type	Description
metadata_spo_site_library_item_id	Edm.String	The combination key of site ID, library ID, and item ID, which uniquely identifies an item in a document library for a site.
metadata_spo_site_id	Edm.String	The ID of the SharePoint site.
metadata_spo_library_id	Edm.String	The ID of document library.
metadata_spo_item_id	Edm.String	The ID of the (document) item in the library.
metadata_spo_item_last_modified	Edm.DateTimeOffset	The last modified date/time (UTC) of the item.
metadata_spo_item_name	Edm.String	The name of the item.
metadata_spo_item_size	Edm.Int64	The size (in bytes) of the item.
metadata_spo_item_content_type	Edm.String	The content type of the item.
metadata_spo_item_extension	Edm.String	The extension of the item.
metadata_spo_item_weburi	Edm.String	The URI of the item.
metadata_spo_item_path	Edm.String	The combination of the parent path and item name.

The SharePoint in Microsoft 365 indexer also supports metadata specific to each document type. More information can be found in [Content metadata properties used in Azure AI Search](#).

① Note

To index custom metadata, "additionalColumns" must be specified in the [query parameter of the data source](#).

Include or exclude by file type

You can control which files are indexed by setting inclusion and exclusion criteria in the "parameters" section of the indexer definition.

Include specific file extensions by setting "`indexedFileNameExtensions`" to a comma-separated list of file extensions (with a leading dot). Exclude specific file extensions by setting "`excludedFileNameExtensions`" to the extensions that should be skipped. If the same extension is in both lists, it's excluded from indexing.

HTTP

```
PUT /indexers/[indexer name]?api-version=2025-11-01-preview
{
    "parameters" : {
        "configuration" : {
            "indexedFileNameExtensions" : ".pdf, .docx",
            "excludedFileNameExtensions" : ".png, .jpeg"
        }
    }
}
```

Controlling which documents are indexed

A single SharePoint in Microsoft 365 indexer can index content from one or more document libraries. To specify which sites and document libraries to index, use the "container" parameter in the data source definition.

The [data source "container" section](#) has two properties for this task: "name" and "query".

Name

The "name" property is required and must be one of three values:

 Expand table

Value	Description
defaultSiteLibrary	Index all content from the site's default document library.

Value	Description
allSiteLibraries	Index all content from all document libraries in a site. Document libraries from a subsite are out of scope/ If you need content from subsites, choose "useQuery" and specify "includeLibrariesInSite".
useQuery	Only index the content defined in the "query".

Query

The "query" parameter of the data source is made up of keyword/value pairs. The below are the keywords that can be used. The values are either site URLs or document library URLs.

① Note

To get the value for a particular keyword, we recommend navigating to the document library that you're trying to include/exclude and copying the URI from the browser. This is the easiest way to get the value to use with a keyword in the query.

[Expand table](#)

Keyword	Value description and examples
null	If null or empty, index either the default document library or all document libraries depending on the container name. Example: <pre>"container" : { "name" : "defaultSiteLibrary", "query" : null }</pre>
includeLibrariesInSite	Index content from all libraries under the specified site in the connection string. The value should be the URI of the site or subsite. Example 1: <pre>"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mycompany.sharepoint.com/mysite" }</pre> Example 2 (include a few subsites only): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mycompany.sharepoint.com/sites/TopSite/SubSite1;includeLibrariesInSite=https://mycompany.sharepoint.com/sites/TopSite/SubSite2" }</pre>
includeLibrary	Index all content from this library. The value is the fully qualified path to the library, which can be copied from your browser. Example 1 (fully qualified path): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/mysite/MyDocumentLibrary" }</pre> Example 2 (URI copied from your browser): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/AllI...</pre>
excludeLibrary	Don't index content from this library. The value is the fully qualified path to the library, which can be copied from your browser. Example 1 (fully qualified path): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mysite.sharepoint.com/subsite1;excludeLibrary=https://mysite.sharepoint.com/subsite1/MyDocumentLibrary" }</pre> Example 2 (URI copied from your browser): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mycompany.sharepoint.com/teams/mysite;excludeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/AllItems.aspx" }</pre>
additionalColumns	Index columns from the document library. The value is a comma-separated list of column names you want to index. Use a double backslash to escape commas in column names: Example 1 (additionalColumns=MyCustomColumn,MyCustomColumn2): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/mysite/MyDocumentLibrary;additionalColumns=MyCustomColumn,MyCustomColumn2" }</pre> Example 2 (escape characters using double backslash): <pre>"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/mysite/MyDocumentLibrary;additionalColumns=MyCustomColumn,MyCustomColumn2" }</pre>

Keyword	Value description and examples
	"includeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/AllItems.aspx;additionalColumns=MyCustomColumnWith\\n\\}

Handling errors

By default, the SharePoint in Microsoft 365 indexer stops as soon as it encounters a document with an unsupported content type (for example, an image). You can use the `excludedFileNameExtensions` parameter to skip certain content types. However, you might need to index documents without knowing all the possible content types in advance. To continue indexing when an unsupported content type is encountered, set the `failOnUnsupportedContentType` configuration parameter to false:

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2025-11-01-preview
Content-Type: application/json
api-key: [admin key]

{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "failOnUnsupportedContentType" : false } }
}
```

For some documents, Azure AI Search is unable to determine the content type, or unable to process a document of otherwise supported content type. To ignore this failure mode, set the `failOnUnprocessableDocument` configuration parameter to false:

HTTP

```
"parameters" : { "configuration" : { "failOnUnprocessableDocument" : false } }
```

Azure AI Search limits the size of documents that are indexed. These limits are documented in [Service Limits in Azure AI Search](#). Oversized documents are treated as errors by default. However, you can still index storage metadata of oversized documents if you set `indexStorageMetadataOnlyForOversizedDocuments` configuration parameter to true:

HTTP

```
"parameters" : { "configuration" : { "indexStorageMetadataOnlyForOversizedDocuments" : true } }
```

You can also continue indexing if errors happen at any point of processing, either while parsing documents or while adding documents to an index. To ignore a specific number of errors, set the `maxFailedItems` and `maxFailedItemsPerBatch` configuration parameters to the desired values. For example:

HTTP

```
{  
    ... other parts of indexer definition  
    "parameters" : { "maxFailedItems" : 10, "maxFailedItemsPerBatch" : 10 }  
}
```

If a file on the SharePoint site has encryption enabled, you might see the following error message:

Code: resourceModified Message: The resource has changed since the caller last read it; usually an eTag mismatch Inner error: Code: irmEncryptFailedToFindProtector

The error message will also include the SharePoint site ID, drive ID, and drive item ID in the following pattern: <sharepoint site id> :: <drive id> :: <drive item id>. This information can be used to identify which item is failing on the SharePoint end. The user can then remove the encryption from the item to resolve the issue.

See also

- YouTube video: SharePoint in Microsoft 365 indexer ↗
 - Indexers in Azure AI Search
 - Content metadata properties used in Azure AI Search
 - Index SharePoint content and other sources in Azure AI Search using Azure Logic App connectors

- [Ingest SharePoint ACL configuration \(preview\)](#)
 - [Preserve and honor Microsoft Purview sensitivity labels \(preview\)](#)
-

(Last updated on 11/18/2025)

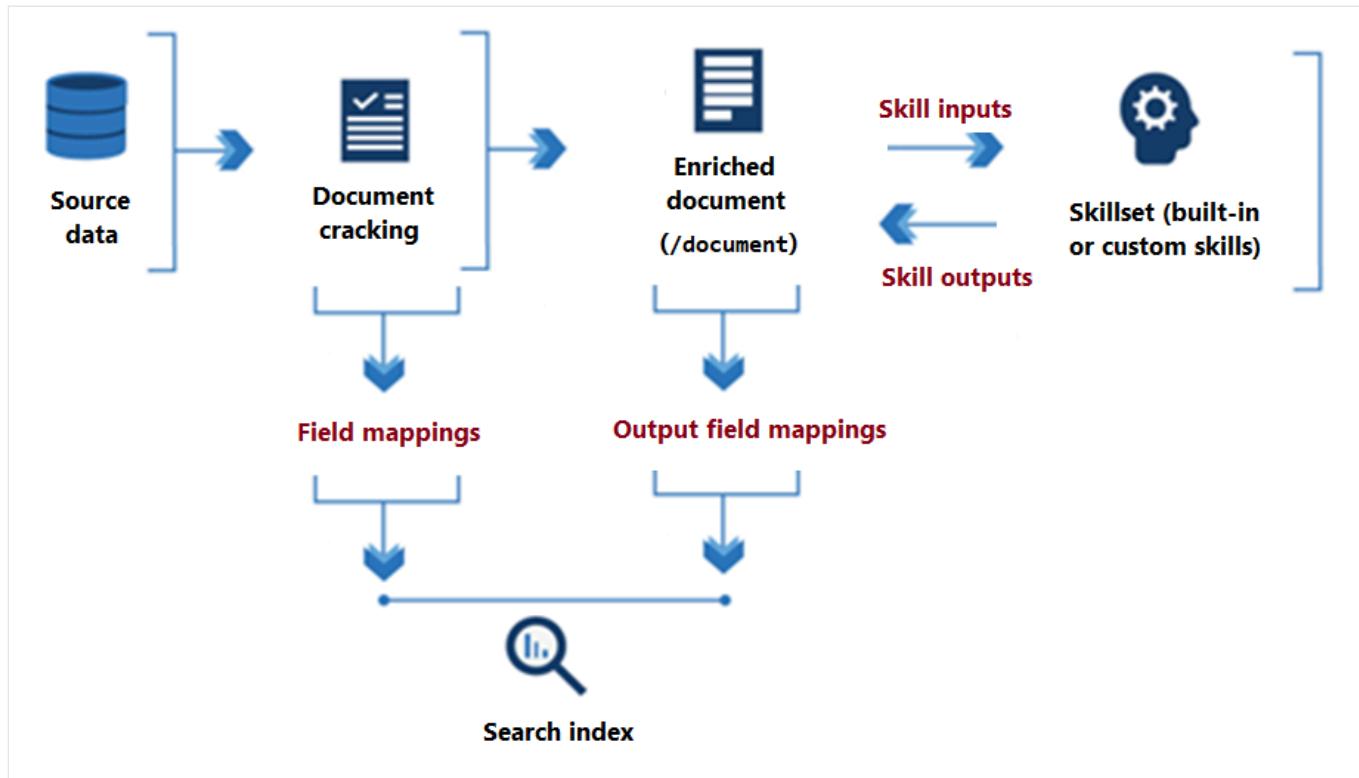
Skillset concepts in Azure AI Search

07/11/2025

This article is for developers who need a deeper understanding of skillset composition, and assumes familiarity with the high-level concepts of [AI enrichment](#), or applied AI, in Azure AI Search.

A skillset is a reusable object in Azure AI Search that's attached to [an indexer](#). It contains one or more skills that call built-in AI or external custom processing over documents retrieved from an external data source.

The following diagram illustrates the basic data flow of skillset execution.



From the onset of skillset processing to its conclusion, skills read from and write to an [enriched document tree](#) that exists in memory. Initially, an enriched document is just the raw content extracted from a data source (articulated as the `"/document"` root node). With each skill execution, the enriched document gains structure and substance as each skill writes its output as nodes in the graph.

After skillset execution is done, the output of an enriched document is routed to an index through user-defined *output field mappings*. Any raw content that you want transferred intact, from source to an index, is defined through *field mappings*. In contrast, *output field mappings* transfer in-memory content (nodes) to the index.

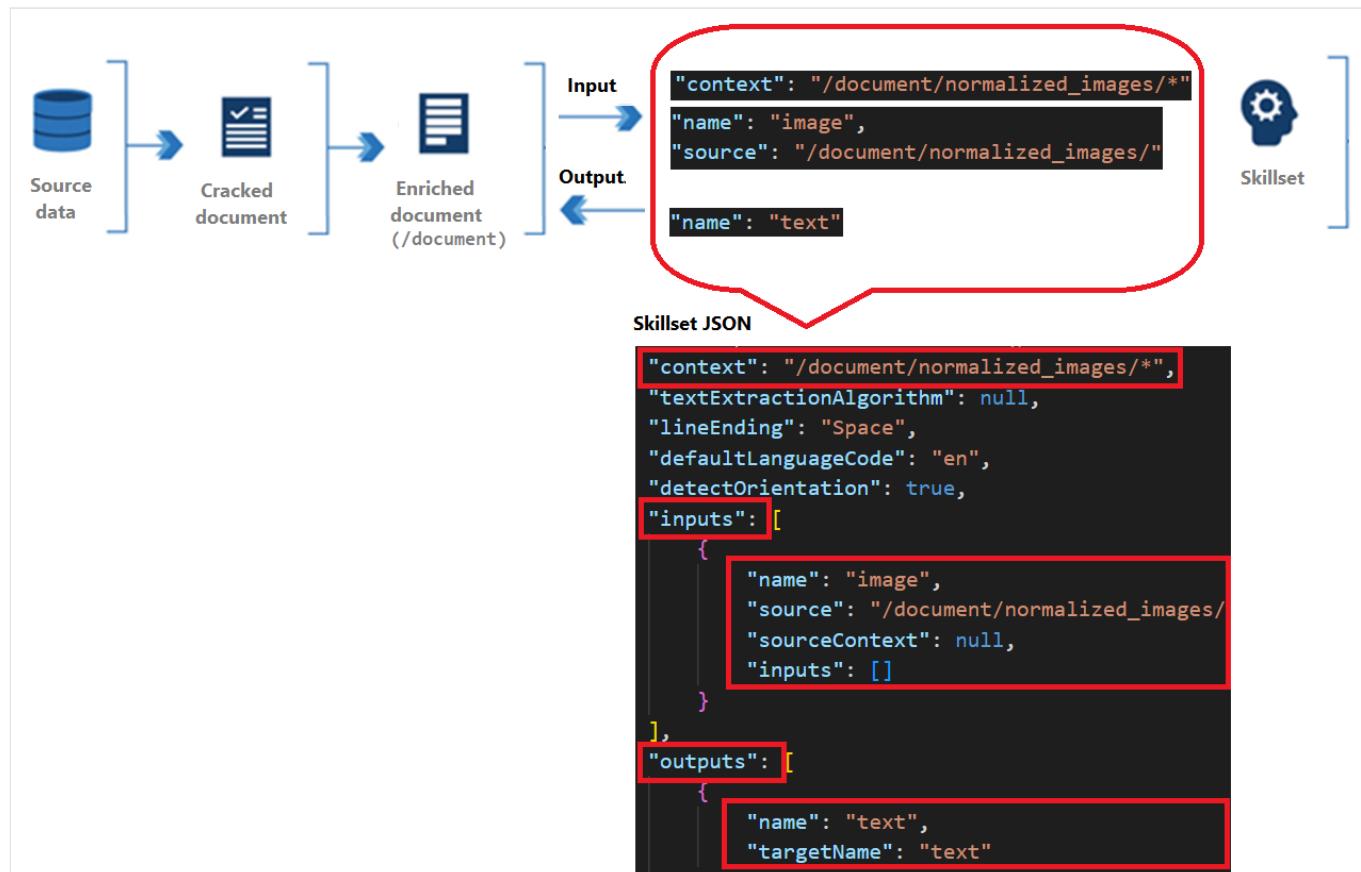
To configure applied AI, specify settings in a skillset and indexer.

Skillset definition

A skillset is an array of one or more *skills* that perform an enrichment, such as translating text or optical character recognition (OCR) on an image file. Skills can be the [built-in skills](#) from Microsoft, or [custom](#)

skills for processing logic that you host externally. A skillset produces enriched documents that are either consumed during indexing or projected to a knowledge store.

Skills have a context, inputs, and outputs:



- **Context** refers to the scope of the operation, which could be once per document or once for each item in a collection.
- Inputs originate from nodes in an enriched document, where a "source" and "name" identify a given node.
- Output is sent back to the enriched document as a new node. Values are the node "name" and node content. If a node name is duplicated, you can set a target name for disambiguation.

Skill context

Each skill has a context, which can be the entire document (`/document`) or a node lower in the tree (`/document/countries/*`).

A context determines:

- The number of times the skill executes, over a single value (once per field, per document), or for a collection, where adding an `/*` results in skill invocation for each instance in the collection.
- Output declaration, or where in the enrichment tree the skill outputs are added. Outputs are always added to the tree as children of the context node.

- Shape of the inputs. For multi-level collections, setting the context to the parent collection affects the shape of the input for the skill. For example if you have an enrichment tree with a list of countries/regions, each enriched with a list of states containing a list of ZIP codes, how you set the context determines how the input is interpreted.

[\[+\] Expand table](#)

Context	Input	Shape of Input	Skill Invocation
/document/countries/*	/document/countries/*/states/*/zipcodes/*	A list of all ZIP codes in the country/region	Once per country/region
/document/countries/*/states/*	/document/countries/*/states/*/zipcodes/*	A list of ZIP codes in the state	Once per combination of country/region and state

Skill dependencies

Skills can execute independently and in parallel, or sequentially in a dependent relationship if you feed the output of one skill into another skill. The following example demonstrates two [built-in skills](#) that execute in sequence:

- Skill #1 is a [Text Split skill](#) that accepts the contents of the "reviews_text" source field as input, and splits that content into "pages" of 5,000 characters as output. Splitting large text into smaller chunks can produce better outcomes for skills like sentiment detection.
- Skill #2 is a [Sentiment Detection skill](#) depends on the split skill output. It accepts "pages" as input, and produces a new field called "Sentiment" as output that contains the results of sentiment analysis.

Notice how the output of the first skill ("pages") is used in sentiment analysis, where "/document/reviews_text/pages/*" is both the context and input. For more information about path formulation, see [How to reference enrichments](#).

```
JSON
{
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
      "name": "#1",
      "description": null,
      "context": "/document/reviews_text",
      "defaultLanguageCode": "en",
      "textSplitMode": "pages",
      "maximumPageLength": 5000,
      "inputs": [
        {
          "id": "#1"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
      "name": "#2",
      "description": null,
      "context": "/document/reviews_text/pages/*",
      "defaultLanguageCode": "en",
      "sentimentType": "TextClassification"
    }
  ]
}
```

```

        {
            "name": "text",
            "source": "/document/reviews_text"
        }
    ],
    "outputs": [
        {
            "name": "textItems",
            "targetName": "pages"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
    "name": "#2",
    "description": null,
    "context": "/document/reviews_text/pages/*",
    "defaultLanguageCode": "en",
    "inputs": [
        {
            "name": "text",
            "source": "/document/reviews_text/pages/*",
        }
    ],
    "outputs": [
        {
            "name": "sentiment",
            "targetName": "sentiment"
        },
        {
            "name": "confidenceScores",
            "targetName": "confidenceScores"
        },
        {
            "name": "sentences",
            "targetName": "sentences"
        }
    ]
}
...
]
}

```

Enrichment tree

An enriched document is a temporary, tree-like data structure created during skillset execution that collects all of the changes introduced through skills. Collectively, enrichments are represented as a hierarchy of addressable nodes. Nodes also include any unenriched fields that are passed in verbatim from the external data source. The best approach for examining the structure and content of an enrichment tree is through a [debug session](#) in the Azure portal.

An enriched document exists for the duration of skillset execution, but can be [cached](#) or sent to a [knowledge store](#).

Initially, an enriched document is simply the content extracted from a data source during [document cracking](#), where text and images are extracted from the source and made available for language or image analysis.

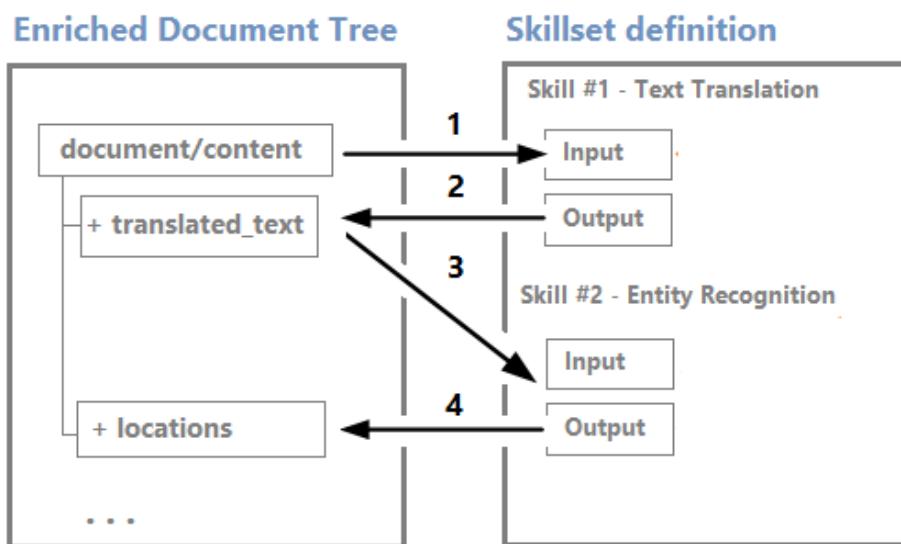
The initial content is metadata and the *root node* (`document/content`). The root node is usually a whole document or a normalized image that is extracted from a data source during document cracking. How it's articulated in an enrichment tree varies for each data source type. The following table shows the state of a document entering into the enrichment pipeline for several supported data sources:

[\[\] Expand table](#)

Data Source\Parsing Mode	Default	JSON, JSON Lines & CSV
Blob Storage	/document/content /document/normalized_images/* ...	/document/{key1} /document/{key2} ...
Azure SQL	/document/{column1} /document/{column2} ...	N/A
Azure Cosmos DB	/document/{key1} /document/{key2} ...	N/A

As skills execute, output is added to the enrichment tree as new nodes. If skill execution is over the entire document, nodes are added at the first level under the root.

Nodes can be used as inputs for downstream skills. For example, skills that create content, such as translated strings, could become input for skills that recognize entities or extract key phrases.



Although you can visualize and work with an enrichment tree through the [Debug Sessions visual editor](#), it's mostly an internal structure.

Enrichments are immutable: once created, nodes can't be edited. As your skillsets get more complex, so will your enrichment tree, but not all nodes in the enrichment tree need to make it to the index or the knowledge store.

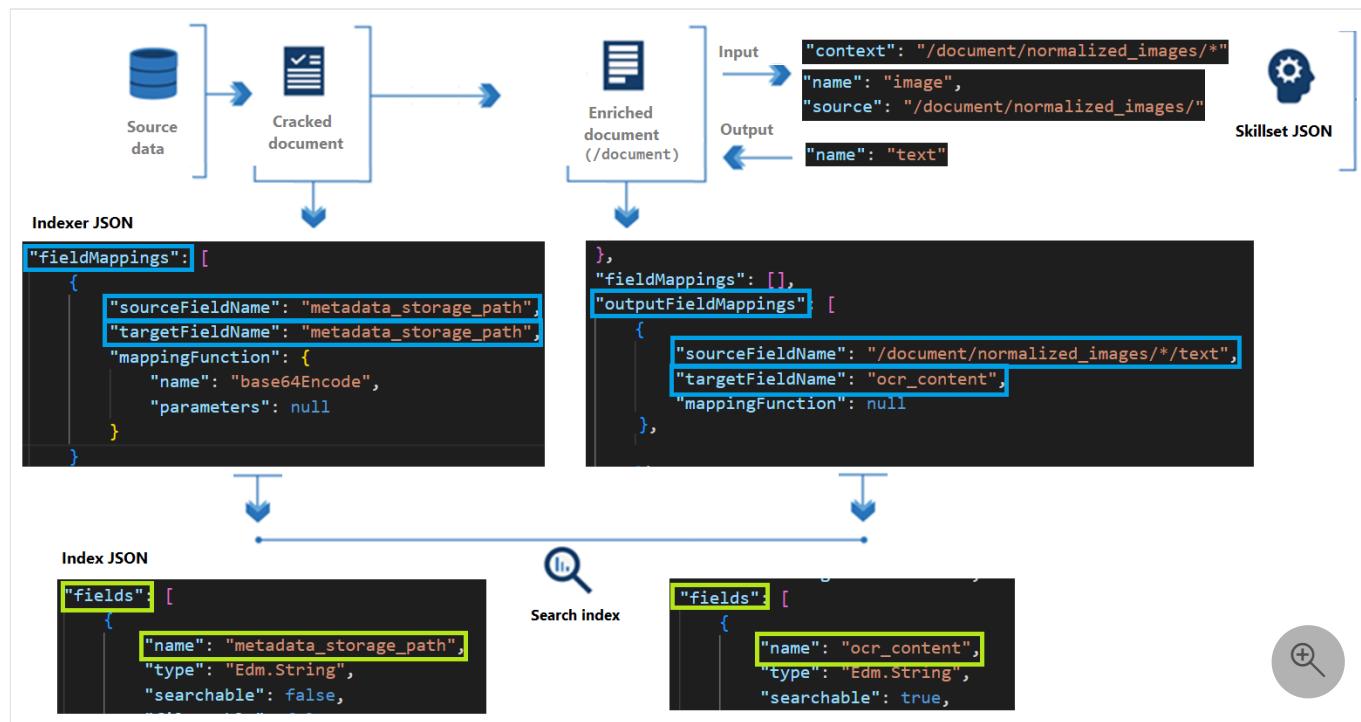
You can selectively persist just a subset of the enrichment outputs so that you're only keeping what you intend to use. The [output field mappings](#) in your indexer definition determines what content actually gets ingested in the search index. Likewise, if you're creating a knowledge store, you can map outputs into [shapes](#) that are assigned to projections.

ⓘ Note

The enrichment tree format enables the enrichment pipeline to attach metadata to even primitive data types. The metadata won't be a valid JSON object, but can be projected into a valid JSON format in projection definitions in a knowledge store. For more information, see [Shaper skill](#).

Indexer definition

An indexer has properties and parameters used to configure indexer execution. Among those properties are mappings that set the data path to fields in a search index.



There are two sets of mappings:

- "["fieldMappings"](#)" map a source field to a search field.
- "["outputFieldMappings"](#)" map a node in an enriched document to a search field.

The "sourceFieldName" property specifies either a field in your data source or a node in an enrichment tree. The "targetFieldName" property specifies the search field in an index that receives the content.

Enrichment example

Using the [hotel reviews skillset](#) as a reference point, this example explains how an [enrichment tree](#) evolves through skill execution using conceptual diagrams.

This example also shows:

- How a skill's context and inputs work to determine how many times a skill executes
- What the shape of the input is based on the context

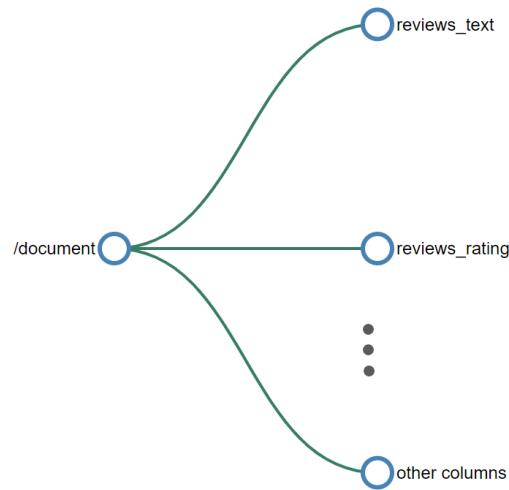
In this example, source fields from a CSV file include customer reviews about hotels ("reviews_text") and ratings ("reviews_rating"). The indexer adds metadata fields from Blob storage, and skills add translated text, sentiment scores, and key phrase detection.

In the hotel reviews example, a "document" within the enrichment process represents a single hotel review.

Tip

You can create a search index and knowledge store for this data in [Azure portal](#) or [REST APIs](#). You can also use [Debug Sessions](#) for insights into skillset composition, dependencies, and effects on an enrichment tree. Images in this article are pulled from Debug Sessions.

Conceptually, the initial enrichment tree looks as follows:



The root node for all enrichments is `"/document"`. When you're working with blob indexers, the `"/document"` node has child nodes of `"/document/content"` and `"/document/normalized_images"`. When the data is CSV, as in this example, the column names map to nodes beneath `"/document"`.

Skill #1: Split skill

When source content consists of large chunks of text, it's helpful to break it into smaller components for [integrated vectorization](#), or for greater accuracy of language, sentiment, and key phrase detection. There are two grains available: pages and sentences. A page consists of approximately 5,000 characters.

An alternative to chunking with the Split skill is through the [Document Layout skill](#), but that skill is out of scope for this article.

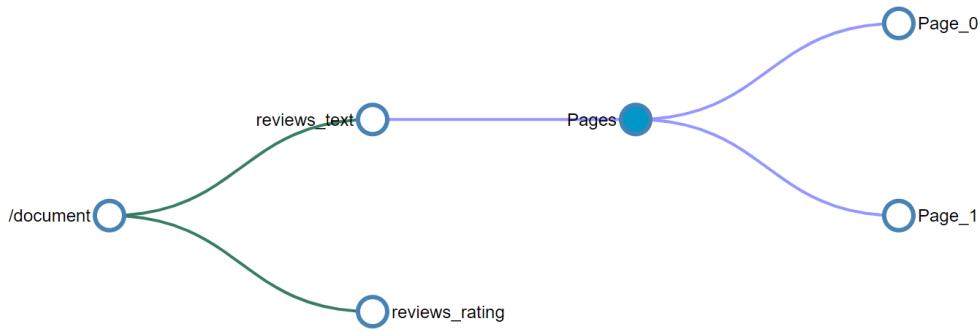
When chunking is required, the Split skill is typically first in a skillset.

```
JSON

{@odata.type": "#Microsoft.Skills.Text.SplitSkill",
"name": "#1",
"description": null,
"context": "/document/reviews_text",
"defaultLanguageCode": "en",
"textSplitMode": "pages",
"maximumPageLength": 5000,
"inputs": [
  {
    "name": "text",
    "source": "/document/reviews_text"
  }
],
"outputs": [
  {
    "name": "textItems",
    "targetName": "pages"
  }
]
```

With the skill context of `"/document/reviews_text"`, the split skill executes once for the `reviews_text`. The skill output is a list where the `reviews_text` is chunked into 5,000 character segments. The output from the split skill is named `pages` and it's added to the enrichment tree. The `targetName` feature allows you to rename a skill output before being added to the enrichment tree.

The enrichment tree now has a new node placed under the context of the skill. This node is available to any skill, projection, or output field mapping.

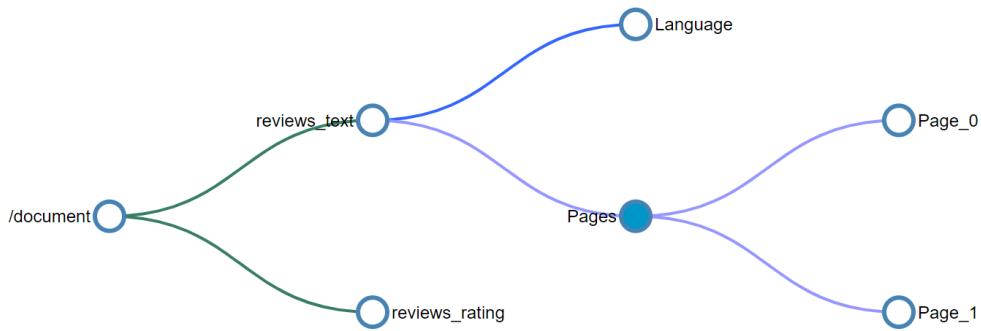


To access any of the enrichments added to a node by a skill, the full path for the enrichment is needed. For example, if you want to use the text from the `pages` node as an input to another skill, specify it as `"/document/reviews_text/pages/*"`. For more information about paths, see [Reference enrichments](#).

Skill #2 Language detection

Hotel review documents include customer feedback expressed in multiple languages. The language detection skill determines which language is used. The result will then be passed to key phrase extraction and sentiment detection (not shown), taking language into consideration when detecting sentiment and phrases.

While the language detection skill is the third (skill #3) skill defined in the skillset, it's the next skill to execute. It doesn't require any inputs so it executes in parallel with the previous skill. Like the split skill that preceded it, the language detection skill is also invoked once for each document. The enrichment tree now has a new node for language.

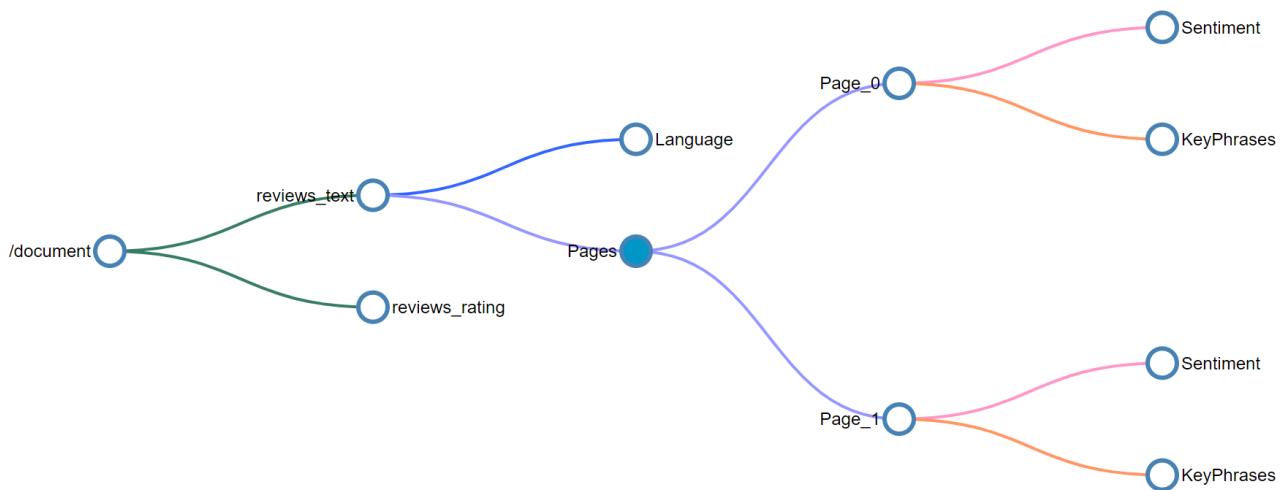


Skills #3 and #4 (sentiment analysis and key phrase detection)

Customer feedback reflects a range of positive and negative experiences. The sentiment analysis skill analyzes the feedback and assigns a score along a continuum of negative to positive numbers, or neutral if sentiment is undetermined. Parallel to sentiment analysis, key phrase detection identifies and extracts words and short phrases that appear consequential.

Given the context of `/document/reviews_text/pages/*`, both sentiment analysis and key phrase skills are invoked once for each of the items in the `pages` collection. The output from the skill will be a node under the associated page element.

You should now be able to look at the rest of the skills in the skillset and visualize how the enrichment tree grows with the execution of each skill. Some skills, such as the merge skill and the shaper skill, also create new nodes but only use data from existing nodes and don't create net new enrichments.



The colors of the connectors in the tree above indicate that the enrichments were created by different skills and the nodes need to be addressed individually and won't be part of the object returned when selecting the parent node.

Skill #5 Shaper skill

If output includes a [knowledge store](#), add a [Shaper skill](#) as a last step. The Shaper skill creates data shapes out of nodes in an enrichment tree. For example, you might want to consolidate multiple nodes into a single shape. You can then project this shape as a table (nodes become the columns in a table), passing the shape by name to a table projection.

The Shaper skill is easy to work with because it focuses shaping under one skill. Alternatively, you can opt for in-line shaping within individual projections. The Shaper Skill doesn't add or detract from an enrichment tree, so it's not visualized. Instead, you can think of a Shaper skill as the means by which you rearticulate the enrichment tree you already have. Conceptually, this is similar to creating views out of tables in a database.

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",  
  "name": "#5",  
  "description": null,  
  "context": "/document",  
  "inputs": [  
    {  
      "name": "name",  
      "source": "/document/name"  
    },  
    {  
      "name": "reviews_date",  
      "source": "/document/reviews_date"  
    },  
    {  
      "name": "reviews_rating",  
      "source": "/document/reviews_rating"  
    },  
    {  
      "name": "reviews_text",  
      "source": "/document/reviews_text"  
    },  
    {  
      "name": "reviews_title",  
      "source": "/document/reviews_title"  
    },  
    {  
      "name": "AzureSearch_DocumentKey",  
      "source": "/document/AzureSearch_DocumentKey"  
    },  
    {  
      "name": "pages",  
      "sourceContext": "/document/reviews_text/pages/*",  
      "inputs": [  
        {  
          "name": "Sentiment",  
          "source": "/document/reviews_text/pages/*/Sentiment"  
        },  
        {  
          "name": "LanguageCode",  
          "source": "/document/Language"  
        },  
        {  
          "name": "Page",  
          "source": "/document/reviews_text/pages/*"  
        },  
        {  
          "name": "keyphrase",  
          "sourceContext": "/document/reviews_text/pages/*/Keyphrases/*",  
          "inputs": [  
            {  
              "name": "Keyphrases",  
              "source": "/document/reviews_text/pages/*/Keyphrases/*"  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```
        },
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "tableprojection"
        }
    ]
}
```

Next steps

With an introduction and example behind you, try creating your first skillset using [built-in skills](#).

[Create your first skillset](#)

Create a skillset in Azure AI Search

10/03/2025



A skillset defines operations that generate vector and textual content and structure from documents that contain images or raw content. Examples are chunking skills, embedding (vectorization) skills, image verbalization, and other processes like optical character recognition (OCR) for images, entity recognition for undifferentiated text, and text translation. A skillset executes after raw content is extracted from an external data source, and after [field mappings](#) are processed.

This article explains how to create a skillset using [REST APIs](#), but the same concepts and steps apply to other programming languages.

Rules for skillset definition include:

- Must have a unique name within the skillset collection. A skillset is a top-level resource that can be used by any indexer.
- Must have at least one skill. Three to five skills are typical. The maximum is 30.
- A skillset can repeat skills of the same type. For example, a skillset can have multiple Shaper skills.
- A skillset supports chained operations, looping, and branching.

A skillset is attached to an indexer. To use the skillset, reference it in an [indexer](#) and then run the indexer to import data, invoke skills processing, and send output to an [index](#). A skillset is a high-level resource, but it's operational only within indexer processing. As a high-level resource, you can reference it in multiple indexers.

💡 Tip

Enable [enrichment caching](#) to reuse the content you've already processed and lower the cost of development.

Add a skillset definition

Creating a skillset adds it to your search service. Updating a skillset fully overwrites an existing skillset with the contents of the request payload. A best practice for updates is to retrieve the

skillset definition with a GET, modify it, and then update with PUT.

Start with the basic structure. In the [Create Skillset REST API](#), the body of the request is authored in JSON and has the following sections:

JSON

```
{  
    "name": "skillset-template",  
    "description": "A description makes the skillset self-documenting (comments  
aren't allowed in JSON itself)",  
    "skills": [  
        ],  
    "cognitiveServices": {  
        "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",  
        "description": "An Azure AI services resource in the same region as Azure AI  
Search",  
        "key": "<Your-Cognitive-Services-Multi-Service-Key>"  
    },  
    "knowledgeStore": {  
        "storageConnectionString": "<Your-Azure-Storage-Connection-String>",  
        "projections": [  
            {  
                "tables": [ ],  
                "objects": [ ],  
                "files": [ ]  
            }  
        ]  
    },  
    "encryptionKey": { }  
}
```

After the name and description, a skillset has four main properties:

- `skills` array, an unordered [collection of skills](#). Skills are either standalone or chained together through input-output associations, where the output of one transform becomes input to another. Skills can be utilitarian (like splitting text), transformational (based on AI from Azure OpenAI or Azure AI services), or custom skills that you provide. An example of a skills array is provided in the next section.
- `cognitiveServices` is used for [billable skills](#) that call Azure AI services APIs. Remove this section if you aren't using billable skills or Custom Entity Lookup. If you are, attach [an Azure AI services multi-service resource](#).
- `knowledgeStore` (optional) specifies an Azure Storage account and settings for projecting skillset output into tables, blobs, and files in Azure Storage. Remove this section if you don't need it, otherwise [specify a knowledge store](#).

- `encryptionKey` (optional) specifies an Azure Key Vault and [customer-managed keys](#) used to encrypt sensitive content (descriptions, connection strings, keys) in a skillset definition. Remove this property if you aren't using customer-managed encryption.

Add skills

Inside the skillset definition, the `skills` array specifies which skills to execute. Three to five skills are common, but you can add as many skills as necessary, subject to [service limits](#).

The end result of an enrichment pipeline is textual content in either a search index or a knowledge store. For this reason, most skills either create text from images (OCR text, captions, tags), or analyze existing text to create new information (entities, key phrases, sentiment). Skills that operate independently are processed in parallel. Skills that depend on each other specify the output of one skill (such as key phrases) as the input of a second skill (such as text translation). The search service determines the order of skill execution and the [execution environment](#).

All skills have a type, context, inputs, and outputs. A skill might optionally have a name and description. The following example shows two unrelated [built-in skills](#) so that you can compare the basic structure.

JSON

```
"skills": [
  {
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "name": "#1",
    "description": "This skill detects organizations in the source content",
    "context": "/document",
    "categories": [
      "Organization"
    ],
    "inputs": [
      {
        "name": "text",
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "organizations",
        "targetName": "orgs"
      }
    ]
  },
  {
    "name": "#2",
    "description": "This skill detects corporate logos in the source files",
  }
]
```

```
 "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
 "context": "/document/normalized_images/*",
 "visualFeatures": [
     "brands"
 ],
 "inputs": [
     {
         "name": "image",
         "source": "/document/normalized_images/*"
     }
 ],
 "outputs": [
     {
         "name": "brands"
     }
 ]
}
```

Each skill is unique in terms of its input values and the parameters that it takes. [Skill reference documentation](#) describes all of the parameters and properties of a given skill. Although there are differences, most skills share a common set and are similarly patterned.

➊ Note

You can build complex skillsets with looping and branching using the [Conditional cognitive skill](#) to create the expressions. The syntax is based on the [JSON Pointer](#) path notation, with a few modifications to identify nodes in the enrichment tree. A `"/"` traverses a level lower in the tree and `"*"` acts as a for-each operator in the context. Numerous examples in this article illustrate [the syntax](#).

Set skill context

Each skill has a [context property](#) that determines the level at which operations take place. If the `context` property isn't explicitly set, the default is `"/document"`, where the context is the whole document (the skill is called once per document).

JSON

```
"skills": [
    {
        "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
        "context": "/document",
        "inputs": [],
        "outputs": []
    },
}
```

```
{
  "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
  "context": "/document/normalized_images/*",
  "visualFeatures": [],
  "inputs": [],
  "outputs": []
}
```

The `context` property is usually set to one of the following examples:

[] [Expand table](#)

Context example	Description
<code>context: /document</code>	(Default) Inputs and outputs are at the document level.
<code>context: /document/pages/*</code>	Some skills like sentiment analysis perform better over smaller chunks of text. If you're splitting a large content field into pages or sentences, the context should be over each component part.
<code>context: /document/normalized_images/*</code>	For image content, inputs and outputs are one per image in the parent document.

Context also determines where outputs are produced in the [enrichment tree](#). For example, the Entity Recognition skill returns a property called `organizations`, captured as `orgs`. If the context is `"/document"`, then an `organizations` node is added as a child of `"/document"`. If you then want to reference this node in downstream skills, the path is `"/document/orgs"`.

Define inputs

Skills read from and write to an enriched document. Skill inputs specify the origin of the incoming data. It's often the root node of the enriched document. For blobs, a typical skill input is the document's content property.

[Skill reference documentation](#) for each skill describes the inputs it can consume. Each input has a `name` that identifies a specific input, and a `source` that specifies the location of the data in the enriched document. The following example is from the Entity Recognition skill:

JSON

```
"inputs": [
  {
    "name": "text",
    "source": "/document/content"
  },
  {
    "name": "language",
    "source": "/document/language"
  }
]
```

```
{
  "name": "languageCode",
  "source": "/document/language"
}
]
```

- Skills can have multiple inputs. The `name` is the specific input. For Entity Recognition, the specific inputs are `text` and `languageCode`.
- The `source` property specifies which field or row provides the content to be processed. For text-based skills, the source is a field in the document or row that provides text. For image-based skills, the node providing the input is normalized images.

 Expand table

Source example	Description
<code>source: /document</code>	For a tabular data set, a document corresponds to a row.
<code>source: /document/content</code>	For blobs, the source is usually the blob's content property.
<code>source: /document/some-named-field</code>	For text-based skills, such as entity recognition or key phrase extraction, the origin should be a field that contains sufficient text to be analyzed, such as a <i>description</i> or <i>summary</i> .
<code>source: /document/normalized_images/*</code>	For image content, the source is image that's been normalized during document cracking.

If the skill iterates over an array, both context and input source should include `/*` in the correct positions. For more information about the complete syntax, see [Skill context and input annotation language](#).

Define outputs

Each skill is designed to emit specific kinds of output, which are referenced by name in the skillset. A skill output has a `name` and an optional `targetName`.

[Skill reference documentation](#) for each skill describes the outputs it can produce. The following example is from the Entity Recognition skill:

JSON

```
"outputs": [
  {
    "name": "persons",
    "targetName": "people"
```

```
        },
        {
            "name": "organizations",
            "targetName": "orgs"
        },
        {
            "name": "locations",
            "targetName": "places"
        }
    ]
}
```

- Skills can have multiple outputs. The `name` property identifies a specific output. For example, for Entity Recognition, output can be *persons*, *locations*, *organizations*, among others.
- The `targetName` property specifies the name you would like this node to have in the enriched document. This is useful if skill outputs have the same name. If you have multiple skills that return the same output, use `targetName` for name disambiguation in enrichment node paths. If the target name is unspecified, the name property is used for both.

Some situations call for referencing each element of an array separately. For example, suppose you want to pass *each element* of `"/document/orgs"` separately to another skill. To do so, add an asterisk to the path: `"/document/orgs/*"`.

Skill output is written to the enriched document as a new node in the enrichment tree. It might be a simple value, such as a sentiment score or language code. It could also be a collection, such as a list of organizations, people, or locations. Skill output can also be a complex structure, as is the case with the Shaper skill. The inputs of the skill determine the composition of the shape, but the output is the named object, which can be referenced in a search index, a knowledge store projection, or another skill by its name.

Add a custom skill

This section includes an example of a [custom skill](#). The URI points to an Azure Function, which in turn invokes the model or transformation that you provide. For more information, see [Add a custom skill to an Azure AI Search enrichment pipeline](#).

Although the custom skill executes code that is external to the pipeline, in a skills array, it's just another skill. Like the built-in skills, it has a type, context, inputs, and outputs. It also reads and writes to an enrichment tree, just as the built-in skills do. Notice that the `context` field is set to `"/document/orgs/*"` with an asterisk, meaning the enrichment step is called *for each* organization under `"/document/orgs"`.

Output, such as the company description in this example, is generated for each organization that's identified. When referring to the node in a downstream step (for example, in key phrase extraction), you would use the path `"/document/orgs/*/companyDescription"` to do so.

```
JSON

{
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
    "description": "This skill calls an Azure function, which in turn calls custom code",
    "uri": "https://indexer-e2e-webskill.azurewebsites.net/api/InvokeCode?code=foo",
    "httpHeaders": {
        "Ocp-Apim-Subscription-Key": "foobar"
    },
    "context": "/document/orgs/*",
    "inputs": [
        {
            "name": "query",
            "source": "/document/orgs/*"
        }
    ],
    "outputs": [
        {
            "name": "description",
            "targetName": "companyDescription"
        }
    ]
}
```

Send output to a destination

Although skill output can be optionally cached for reuse purposes, it's usually temporary and exists only while skill execution is in progress.

- To send output to a field in a search index, [create an output field mapping](#) in an indexer.
- To send output to a knowledge store, [create a projection](#).
- To send output to a downstream skill, reference the output by its node name, such as `"/document/organization"`, in the downstream skill's input source property. See [Reference an annotation](#) for examples.

Tips for a first skillset

- Try the [Import data wizard](#) or [Import data \(new\) wizard](#).

The wizards automate several steps that can be challenging the first time around. It defines the skillset, index, and indexer, including field mappings and output field mappings. It also defines projections in a knowledge store if you're using one. For some skills, such as OCR or image analysis, the wizard adds utility skills that merge the image and text content that was separated during document cracking.

After the wizard runs, you can open each object in the Azure portal to view its JSON definition.

- Try [Debug Sessions](#) to invoke skillset execution over a target document and inspect the enriched document that the skillset creates. You can view and modify input and output settings and values. This tutorial is a good place to start: [Tutorial: Debug a skillset using Debug Sessions](#).

Next step

Context and input source fields are paths to nodes in an enrichment tree. As a next step, learn more about the path syntax for nodes in an enrichment tree.

[Referencing annotations in a skillset](#)

Tips for AI enrichment in Azure AI Search

05/08/2025

This article contains tips to help you get started with AI enrichment and skillsets used during indexing.

Tip 1: Start simple and start small

Both the [Import data wizard](#) and [Import and vectorize data wizard](#) in the Azure portal support AI enrichment. Without writing any code, you can create and examine all of the objects used in an enrichment pipeline: an index, indexer, data source, and skillset.

Another way to start simply is by creating a data source with just a handful of documents or rows in a table that are representative of the documents that will be indexed. A small data set is the best way to increase the speed of finding and fixing issues. Run your sample through the end-to-end pipeline and check that the results meet your needs. Once you're satisfied with the results, you're ready to add more files to your data source.

Tip 2: See what works even if there are some failures

Sometimes a small failure stops an indexer in its tracks. That is fine if you plan to fix issues one by one. However, you might want to ignore a particular type of error, allowing the indexer to continue so that you can see what flows are actually working.

To ignore errors during development, set `maxFailedItems` and `maxFailedItemsPerBatch` as -1 as part of the indexer definition.

JSON

```
{  
  // rest of your indexer definition  
  "parameters":  
  {  
    "maxFailedItems": -1,  
    "maxFailedItemsPerBatch": -1  
  }  
}
```

 Note

As a best practice, set the `maxFailedItems` and `maxFailedItemsPerBatch` to 0 for production workloads

Tip 3: Use Debug session to troubleshoot issues

Debug session is a visual editor that shows a skillset's dependency graph, inputs and outputs, and definitions. It works by loading a single document from your search index, with the current indexer and skillset configuration. You can then run the entire skillset, scoped to a single document. Within a debug session, you can identify and resolve errors, validate changes, and commit changes to a parent skillset. For a walkthrough, see [Tutorial: debug sessions](#).

Tip 4: Expected content fails to appear

If you're missing content, check for dropped documents in the Azure portal. In the search service page, open **Indexers** and look at the **Docs succeeded** column. Click through to indexer execution history to review specific errors.

If the problem is related to file size, you might see an error like this: "The blob <file-name>" has the size of <file-size> bytes, which exceed the maximum size for document extraction for your current service tier." For more information on indexer limits, see [Service limits](#).

A second reason for content failing to appear might be related input/output mapping errors. For example, an output target name is "People" but the index field name is lower-case "people". The system could return 201 success messages for the entire pipeline so you think indexing succeeded, when in fact a field is empty.

Tip 5: Extend processing beyond maximum run time

Image analysis is computationally intensive for even simple cases, so when images are especially large or complex, processing times can exceed the maximum time allowed.

For indexers that have skillsets, skillset execution is [capped at 2 hours for most tiers](#). If skillset processing fails to complete within that period, you can put your indexer on a 2-hour recurring schedule to have the indexer pick up processing where it left off.

Scheduled indexing resumes at the last known good document. On a recurring schedule, the indexer can work its way through the image backlog over a series of hours or days, until all unprocessed images are processed. For more information on schedule syntax, see [Schedule an indexer](#).

Note

If an indexer is set to a certain schedule but repeatedly fails on the same document over and over again each time it runs, the indexer will begin running on a less frequent interval (up to the maximum of at least once every 24 hours) until it successfully makes progress again. = If you believe you have fixed whatever the issue that was causing the indexer to be stuck at a certain point, you can perform an on-demand run of the indexer, and if that successfully makes progress, the indexer will return to its set schedule interval again.

Tip 6: Increase indexing throughput

For [parallel indexing](#), distribute your data into multiple containers or multiple virtual folders inside the same container. Then create multiple data source and indexer pairs. All indexers can use the same skillset and write into the same target search index, so your search app doesn't need to be aware of this partitioning.

See also

- [Quickstart: Create an AI enrichment pipeline in the Azure portal](#)
- [Tutorial: Learn AI enrichment REST APIs](#)
- [How to configure blob indexers](#)
- [How to define a skillset](#)
- [How to map enriched fields to an index](#)

Attach a billable resource to a skillset in Azure AI Search

If you're using built-in skills for [AI enrichment](#) in Azure AI Search, you can enrich a small number of documents for free, up to 20 transactions per index per day. For larger or more frequent workloads, you should attach a billable Microsoft Foundry resource to your [skillset](#).

Azure AI Search uses dedicated, internally hosted resources to execute built-in skills backed by Foundry Tools and requires a Foundry resource solely for billing purposes. The exception is the [Azure Content Understanding skill](#), which uses your resource for both billing and processing.

A Foundry resource provides access to multiple services within Foundry Tools. When you specify it in a skillset, Microsoft is able to charge you for using the following services:

- [Azure Vision in Foundry Tools](#) for image analysis, optical character recognition (OCR), and multimodal embeddings.
- [Azure Language in Foundry Tools](#) for language detection, entity recognition, sentiment analysis, and key phrase extraction.
- [Azure Translator in Foundry Tools](#) for machine text translation.

Skillset processing is billed to the underlying service of each skill. Azure AI Search consolidates charges for Foundry Tools into a single Foundry resource. For example, if you use the [Image Analysis](#) and [Language Detection](#) skills, charges for Azure Vision and Azure Language appear on the same bill for your Foundry resource. All other resources are billed independently.

To attach a Foundry resource, provide connection information in the skillset. You can use a key-based approach or keyless approach, which is currently in preview.

Prerequisites

- Connectivity over a public endpoint, unless your search service meets the creation date, tier, and region requirements for [private connections](#) to a Foundry resource.
- A [Foundry resource](#) with the `AI Services` API kind. You can verify the API kind on the resource's [Overview](#) page in the Azure portal:

Endpoints	: Click here to view endpoints
API Kind	: AI Services
Location	:
Status	: Succeeded 

(!) Note

- If your Foundry resource is configured to use a private endpoint, Azure AI Search can [connect using a shared private link](#). For more information, see [Shared private link resource limits](#).
- The 2025-11-01-preview introduces support for the `AI Services` API kind. The previous `CognitiveServices` and classic Azure AI multi-service accounts continue to work, but for new skillsets, we recommend that you use `AI Services` and Foundry resources.

Bill through a keyless connection

(!) Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

You can use a managed identity and permissions to attach a Foundry resource. The advantage of this approach is that billing is keyless and doesn't have region requirements.

As with keys, the details you provide about the resource are used for billing, not connections. All API requests made by Azure AI Search to Foundry Tools for built-in skills processing remain internal and managed by Microsoft.

To bill through a keyless connection:

1. On your Azure AI Search service, [configure a managed identity](#). Both system-assigned and user-assigned identities are supported.

2. On your Foundry resource, assign the [Cognitive Services User role](#) to the managed identity of your search service.
3. Configure a skillset to use the managed identity. You can use the Azure portal, the latest preview version of [Skillsets - Create Or Update \(REST API\)](#), or an Azure SDK beta package that provides the syntax.
 - `@odata.type` is always `#Microsoft.Azure.Search.AIServicesByIdentity`.
 - `subdomainUrl` is the endpoint of your Foundry resource. You can use the `https://<resource-name>.services.ai.azure.com` or `https://<resource-name>.cognitiveservices.azure.com` format, both of which are available on the [Keys and Endpoint](#) page in the Azure portal.
 - Other properties are specific to the type of managed identity, as shown in the following REST API examples.

System-assigned managed identity

Here's a sample skillset configuration for a system-assigned managed identity. In this scenario, you must set `identity` to `null`.

HTTP

```
POST https://[service-name].search.windows.net/skillsets/[skillset-name]?api-version=2025-11-01-preview
api-key: [admin-key]
Content-Type: application/json

{
    "name": "my-skillset",
    "skills": [
        // Skills definition goes here
    ],
    "cognitiveServices": {
        "@odata.type": "#Microsoft.Azure.Search.AIServicesByIdentity",
        "description": "A sample configuration for a system-assigned managed identity.",
        "subdomainUrl": "https://[resource-name].services.ai.azure.com",
        "identity": null
    }
}
```

Bill through a resource key

By default, Azure AI Search charges for transactions using the key of a Foundry resource. This approach is generally available. You can use the Azure portal, a stable REST API version, or an equivalent Azure SDK to add the key to a skillset.

There are two supported key types:

- `#Microsoft.Azure.Search.CognitiveServicesByKey` calls the regional endpoint.
- `#Microsoft.Azure.Search.AIPropertiesByKey` calls the subdomain. We recommend this type because it supports shared private links and doesn't have regional requirements relative to the search service.

Your Foundry resource must be in the same region as your search service. Choose an [Azure AI Search region that provides Foundry Tools integration](#), which is indicated by the **AI enrichment** column. For more information about the same-region requirement, see [How the key is used](#).

If you don't specify the `cognitiveServices` property, your search service attempts to use the free enrichments available to your indexer each day. Execution of billable skills stops at 20 transactions per indexer invocation, and a "Time Out" message appears in the indexer execution history.

Azure portal

1. Sign in to the [Azure portal](#).
2. Create a Foundry resource in the same region as your search service.
3. From the left pane, select **Resource Management > Keys and Endpoint**.
4. Copy one of the keys.
 - If you're using an [import wizard](#), select the Foundry resource. The wizard adds the resource key to your skillset definition.
 - For a new or existing skillset, provide the key in skillset definition.
5. Add the key to a skillset definition.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a navigation bar with 'Microsoft Azure' at the top, followed by 'Home > contoso-search-eastus | Skillsets > py-rag-tutorial-ss ...'. Below this is a toolbar with 'Save', 'Refresh', 'Delete', and 'Connect AI service' (which is highlighted with a red box). The main area is titled 'Skillset Definition (JSON)' and shows a code editor with JSON code. A specific section of the code, which defines 'cognitiveServices', is also highlighted with a red box. On the right, a modal window titled 'Connect AI service' is open. It contains a table where an AI services resource named 'contosoazureaimultiservice' is listed under the 'Free (Limited enrichments)' plan in the 'eastus' region. There are 'Save' and 'Cancel' buttons at the bottom of the modal.

➊ Note

The portal automatically attaches the key of type
#Microsoft.Azure.Search.CognitiveServicesByKey .

Remove the key

Enrichments are billable operations. If you no longer need to call Foundry Tools, follow these instructions to remove the key and prevent use of the Foundry resource.

Without the key, the skillset reverts to the default allocation of 20 free transactions per indexer per day. Execution of billable skills stops at 20 transactions, and a "Time Out" message appears in the indexer execution history when the allocation is used.

Azure portal

1. Sign in to the [Azure portal](#).
2. Under **Search management > Skillsets**, select a skillset from the list.

contoso-search-eastus | Skillsets

Search

Add skillset Refresh Delete

Access control (IAM)

Tags

Diagnose and solve problems

Search management

Indexes

Indexers

Data sources

Aliases

Skillsets

Debug sessions

Filter by name...

Name	Number of Skills
py-rag-tutorial-ss	3
text-image-vectors-skillset	5
text-only-vectors-skillset	2
vector-1729263603400-skillset	5

3. Scroll to the "cognitiveServices" section in the file.

4. Delete the key value from the JSON.

5. Save the skillset.

py-rag-tutorial-ss

Skillset

Save Refresh Delete Connect AI service

Skillset Definition (JSON) Knowledge Store

```
77     }
78   ]
79 }
80 ],
81 "cognitiveServices": [
82   "@odata.type": "#Microsoft.Azure.Search.CognitiveSearchSkillDefinitionTemplate",
83   "description": null,
84   "key": "" // This line is highlighted with a red box
85 },
86   "knowledgeStore": null,
87   "indexProjections": {
```

Skill Definition Templates

Skills

Azure Machine Learning (AML)

Integrate a model built in Azure Machine Learning as a skill.

Learn more

Workspaces

Choose a workspace

How the key is used

Billing goes into effect when API calls to a Foundry resource exceed 20 API calls per indexer per day. You can [reset the indexer](#) to reset the API count.

Keyless and key-based connections are used for billing, but not for connections related to enrichment operations.

For key-based connections, a search service [connects over the internal network](#) to a Foundry resource located in the [same physical region](#). Most regions that offer Azure AI Search also offer other Azure services. If you attempt AI enrichment in a region that doesn't have both services, you see this message: "Provided key isn't a valid CognitiveServices type key for the region of your search service."

For keyless connections, a search service authenticates using its identity and role assignment and targets a Foundry resource. The resource is specified as a fully qualified URI, and the URI includes a unique subdomain.

Indexers can be configured to run in a [private execution environment](#) for dedicated processing using just the search nodes of your own search service. Even if you're using a private execution environment, Azure AI Search still uses its internally provisioned resources to perform all skill enrichments.

 **Note**

Some built-in skills, such as the [Text Translation skill](#), are based on non-regional Foundry Tools services. If you use a non-regional skill, your request might be serviced in a different region than the Azure AI Search region. For more information about non-regional services, see the [product availability by region](#) ↗ page.

Public connection requirements

Depending on when your search service was created, its pricing tier, and its region, billing for [built-in skills](#) can require a public connection from Azure AI Search to your Foundry resource. Disabling public network access breaks billing in some scenarios. Review the requirements for [connections through a shared private link](#) to determine whether your search service requires a public connection.

If you can't use the public network, you can configure a [Custom Web API skill](#) implemented with an [Azure Function](#) that supports [private endpoints](#) and add your [Foundry resource to the same VNET](#). In this scenario, you can call your Foundry resource directly from the custom skill using private endpoints.

Key requirements special cases

[Custom Entity Lookup](#) is metered by Azure AI Search, but it requires a Foundry resource key to unlock transactions beyond 20 per indexer per day. For this skill only, the resource key unblocks the number of transactions but is unrelated to billing.

Free enrichments

AI enrichment offers a small quantity of free processing of billable enrichments so that you can complete short exercises without having to attach an external resource. Free enrichments are 20 documents per indexer per day. You can [reset the indexer](#) to reset the counter if you want to repeat an exercise.

Some enrichments are always free:

- Utility skills that don't call Foundry Tools (namely the [Conditional](#), [Document Extraction](#), [Shaper](#), [Text Merge](#), and [Text Split](#) skills) aren't billable.
- Text extraction from PDF documents and other application files is nonbillable. Text extraction, which occurs during [document cracking](#), isn't an AI enrichment, but it occurs during AI enrichment and is thus noted here.

Billable enrichments

During AI enrichment, Azure AI Search calls APIs for [built-in skills](#) that are based on Azure Vision, Azure Language, and Azure Translator.

Billable built-in skills that make backend calls to external services include:

- [Entity Linking](#)
- [Entity Recognition](#)
- [Image Analysis](#)
- [Key Phrase Extraction](#)
- [Language Detection](#)
- [OCR](#)
- [Personally Identifiable Information \(PII\) Detection](#)
- [Sentiment](#)
- [Text Translation](#)
- [Azure Vision multimodal embeddings](#)

A [query-time vectorizer](#) backed by the Azure Vision multimodal embedding model is also a billable enrichment.

Image extraction is an Azure AI Search operation that occurs when documents are cracked prior to enrichment. Image extraction is billable on all pricing tiers, except for 20 free daily extractions on the free tier. Image extraction costs apply to image files inside blobs, embedded images in other files (PDF and other app files), and images extracted using [Document Extraction](#). For image extraction pricing, see the [Azure AI Search pricing page](#).

Tip

To lower the cost of skillset processing, enable [incremental enrichment](#) to cache and reuse any enrichments that are unaffected by changes made to a skillset. Caching requires Azure Storage (see [pricing ↗](#)), but the cumulative cost of skillset execution is lower if existing enrichments can be reused, especially for skillsets that use image extraction and analysis.

Example: Estimate costs

The prices shown in this section are hypothetical and used to illustrate the estimation process. Your costs could be lower. For the actual price of transactions, see [Foundry Tools pricing ↗](#).

To estimate the costs associated with Azure AI Search indexing, start with an idea of what an average document looks like so you can run some numbers. For example, you might approximate:

- 1,000 PDFs
- Six pages each
- One image per page (6,000 images)
- 3,000 characters per page

Assume a pipeline that consists of document cracking of each PDF, image and text extraction, OCR of images, and entity recognition of organizations.

1. For document cracking with text and image content, text extraction is currently free. For 6,000 images, assume \$1 for every 1,000 images extracted. That's a cost of \$6.00 for this step.
2. For OCR of 6,000 images in English, the OCR cognitive skill uses the best algorithm (`DescribeText`). Assuming a cost of \$2.50 per 1,000 images to be analyzed, you would pay \$15.00 for this step.
3. For entity extraction, you'd have a total of three text records per page. Each record is 1,000 characters. Three text records per page multiplied by 6,000 pages equal 18,000 text records. Assuming \$2.00 per 1,000 text records, this step would cost \$36.00.

Putting it all together, you'd pay about \$57.00 to ingest 1,000 PDF documents of this type with the described skillset.

Related content

- [Azure AI Search pricing page ↗](#)
 - [How to define a skillset](#)
 - [Create Skillset \(REST\)](#)
 - [How to map enriched fields](#)
-

Last updated on 11/18/2025

Define an index projection for parent-child indexing

05/08/2025

For indexes containing chunked documents, an *index projection* specifies how parent-child content is mapped to fields in a search index for one-to-many indexing. Through an index projection, you can send content to:

- A single index, where the parent fields repeat for each chunk, but the grain of the index is at the chunk level. The [RAG tutorial](#) is an example of this approach.
- Two or more indexes, where the parent index has fields related to the parent document, and the child index is organized around chunks. The child index is the primary search corpus, but the parent index could be used for [lookup queries](#) when you want to retrieve the parent fields of a particular chunk, or for independent queries.

Most implementations are a single index organized around chunks with parent fields, such as the document filename, repeating for each chunk. However, the system is designed to support separate and multiple child indexes if that's your requirement. Azure AI Search doesn't support index joins so your application code must handle which index to use.

An index projection is defined in a [skillset](#). It's responsible for coordinating the indexing process that sends chunks of content to a search index, along with the parent content associated with each chunk. It improves how native data chunking works by giving you more options for controlling how parent-child content is indexed.

This article explains how to create the index schema and indexer projection patterns for one-to-many indexing.

Prerequisites

- An [indexer-based indexing pipeline](#).
- An index (one or more) that accepts the output of the indexer pipeline.
- A [supported data source](#) having content that you want to chunk. It can be vector or nonvector content.
- A skill that splits content into chunks, either the [Text Split skill](#) or a custom skill that provides equivalent functionality.

The skillset contains the indexer projection that shapes the data for one-to-many indexing. A skillset could also have other skills, such as an embedding skill like [AzureOpenAIEmbedding](#) if your scenario includes integrated vectorization.

Dependency on indexer processing

One-to-many indexing takes a dependency on skillsets and indexer-based indexing that includes the following four components:

- A data source
- One or more indexes for your searchable content
- A skillset that contains an index projection*
- An indexer

Your data can originate from any supported data source, but the assumption is that the content is large enough that you want to chunk it, and the reason for chunking it is that you're implementing a RAG pattern that provides grounding data to a chat model. Or, you're implementing vector search and need to meet the smaller input size requirements of embedding models.

Indexers load indexed data into a predefined index. How you define the schema and whether to use one or more indexes is the first decision to make in a one-to-many indexing scenario. The next section covers index design.

Create an index for one-to-many indexing

Whether you create one index for chunks that repeat parent values, or separate indexes for parent-child field placement, the primary index used for searching is designed around data chunks. It must have the following fields:

- A document key field uniquely identifying each document. It must be defined as type `Edm.String` with the `keyword` analyzer.
- A field associating each chunk with its parent. It must be of type `Edm.String`. It can't be the document key field, and must have `filterable` set to true. It's referred to as `parent_id` in the examples and as a [projected key value](#) in this article.
- Other fields for content, such as text or vectorized chunk fields.

An index must exist on the search service before you create the skillset or run the indexer.

Single index schema inclusive of parent and child fields

A single index designed around chunks with parent content repeating for each chunk is the predominant pattern for RAG and vector search scenarios. The ability to associate the correct parent content with each chunk is enabled through index projections.

The following schema is an example that meets the requirements for index projections. In this example, parent fields are the parent_id and the title. Child fields are the vector and nonvector vector chunks. The chunk_id is the document ID of this index. The parent_id and title repeat for every chunk in the index.

You can use the Azure portal, REST APIs, or an Azure SDK to [create an index](#).

REST

JSON

```
{  
    "name": "my_consolidated_index",  
    "fields": [  
        {"name": "chunk_id", "type": "Edm.String", "key": true, "filterable": true, "analyzer": "keyword"},  
        {"name": "parent_id", "type": "Edm.String", "filterable": true},  
        {"name": "title", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "retrievable": true},  
        {"name": "chunk", "type": "Edm.String", "searchable": true, "retrievable": true},  
        {"name": "chunk_vector", "type": "Collection(Edm.Single)", "searchable": true, "retrievable": false, "stored": false, "dimensions": 1536, "vectorSearchProfile": "hnsw"}  
    ],  
    "vectorSearch": {  
        "algorithms": [{"name": "hnsw", "kind": "hnsw", "hnswParameters": {}}],  
        "profiles": [{"name": "hnsw", "algorithm": "hnsw"}]  
    }  
}
```

Add index projections to a skillset

Index projections are defined inside a skillset definition and are primarily defined as an array of `selectors`, where each selector corresponds to a different target index on the search service. This section starts with syntax and examples for context, followed by [parameter reference](#).

Choose a tab for the various API syntax. There's currently no portal support for setting up projections, other than editing the skillset JSON definition. Refer to the REST example for JSON.

Index projections are generally available. We recommend the most recent stable API:

- [Create Skillset \(api-version=2024-07-01\)](#)

Here's an example payload for an index projections definition that you might use to project individual pages output by the [Text Split skill](#) as their own documents in the search index.

JSON

```
"indexProjections": {  
    "selectors": [  
        {  
            "targetIndexName": "my_consolidated_index",  
            "parentKeyFieldName": "parent_id",  
            "sourceContext": "/document/pages/*",  
            "mappings": [  
                {  
                    "name": "chunk",  
                    "source": "/document/pages/*",  
                    "sourceContext": null,  
                    "inputs": []  
                },  
                {  
                    "name": "chunk_vector",  
                    "source": "/document/pages/*/chunk_vector",  
                    "sourceContext": null,  
                    "inputs": []  
                },  
                {  
                    "name": "title",  
                    "source": "/document/title",  
                    "sourceContext": null,  
                    "inputs": []  
                }  
            ]  
        },  
        {  
            "parameters": {  
                "projectionMode": "skipIndexingParentDocuments"  
            }  
        }  
    ]  
}
```

Parameter reference

 [Expand table](#)

Index	Definition
projection parameters	
<code>selectors</code>	Parameters for the main search corpus, usually the one designed around chunks.
<code>projectionMode</code>	An optional parameter providing instructions to the indexer. The only valid value for this parameter is <code>skipIndexingParentDocuments</code> , and it's used when the chunk index is the primary search corpus and you need to specify whether parent fields are indexed as extra search documents within the chunked index. If you don't set <code>skipIndexingParentDocuments</code> , you get extra search documents in your index that are null for chunks, but populated with parent fields only. For example, if five documents contribute 100 chunks to the index, then the number of documents in the index is 105. The five documents created or parent fields have nulls for chunk (child) fields, making them substantially different from the bulk of the documents in the index. We recommend <code>projectionMode</code> set to <code>skipIndexingParentDocument</code> .

Selectors have the following parameters as part of their definition.

[] [Expand table](#)

Selector parameters	Definition
<code>targetIndexName</code>	The name of the index into which index data is projected. It's either the single chunked index with repeating parent fields, or it's the child index if you're using separate indexes for parent-child content.
<code>parentKeyFieldName</code>	The name of the field providing the key for the parent document.
<code>sourceContext</code>	The enrichment annotation that defines the granularity at which to map data into individual search documents. For more information, see Skill context and input annotation language .
<code>mappings</code>	<p>An array of mappings of enriched data to fields in the search index. Each mapping consists of:</p> <ul style="list-style-type: none"> <code>name</code>: The name of the field in the search index that the data should be indexed into. <code>source</code>: The enrichment annotation path that the data should be pulled from. <p>Each <code>mapping</code> can also recursively define data with an optional <code>sourceContext</code> and <code>inputs</code> field, similar to the knowledge store or Shaper Skill. Depending on your application, these parameters allow you to shape data into fields of type <code>Edm.ComplexType</code> in the search index. Some LLMs don't accept a complex type in search results, so the LLM you're using determines whether a complex type mapping is helpful or not.</p>

The `mappings` parameter is important. You must explicitly map every field in the child index, except for the ID fields such as document key and the parent ID.

This requirement is in contrast with other field mapping conventions in Azure AI Search. For some data source types, the indexer can implicitly map fields based on similar names, or known characteristics (for example, blob indexers use the unique metadata storage path as the default document key). However, for indexer projections, you must explicitly specify every field mapping on the "many" side of the relationship.

Do not create a field mapping for the parent key field. Doing so disrupts change tracking and synchronized data refresh.

Handling parent documents

Now that you've seen several patterns for one-to-many indexings, let's compare key differences about each option. Index projections effectively generate "child" documents for each "parent" document that runs through a skillset. You have several choices for handling the "parent" documents.

- To send parent and child documents to separate indexes, set the `targetIndexName` for your indexer definition to the parent index, and set the `targetIndexName` in the index projection selector to the child index.
- To keep parent and child documents in the same index, set the indexer `targetIndexName` and the index projection `targetIndexName` to the same index.
- To avoid creating parent search documents and ensuring the index contains only child documents of a uniform grain, set the `targetIndexName` for both the indexer definition and the selector to the same index, but add an extra `parameters` object after `selectors`, with a `projectionMode` key set to `skipIndexingParentDocuments`, as shown here:

JSON

```
"indexProjections": {
    "selectors": [
        ...
    ],
    "parameters": {
        "projectionMode": "skipIndexingParentDocuments"
    }
}
```

Review field mappings

Indexers are affiliated with three different types of field mappings. Before you run the indexer, check your field mappings and know when to use each type.

[Field mappings](#) are defined in an indexer and used to map a source field to an index field. Field mappings are used for data paths that lift data from the source and pass it in for indexing, with no intermediate skills processing step. Typically, an indexer can automatically map fields that have the same name and type. Explicit field mappings are only required when there's discrepancies. In one-to-many indexing and the patterns discussed thus far, you might not need field mappings.

[Output field mappings](#) are defined in an indexer and used to map enriched content generated by a skillset to a field into the main index. In the one-to-many patterns covered in this article, this is the parent index in a [two-index solution](#). In the examples shown in this article, the parent index is sparse, with just a title field, and that field isn't populated with content from the skillset processing, so we don't an output field mapping.

Indexer projection field mappings are used to map skillset-generated content to fields in the child index. In cases where the child index also includes parent fields (as in the [consolidated index solution](#)), you should set up field mappings for every field that has content, including the parent-level title field, assuming you want the title to show up in each chunked document. If you're using [separate parent and child indexes](#), the indexer projections should have field mappings for just the child-level fields.

Note

Both output field mappings and indexer projection field mappings accept enriched document tree nodes as source inputs. Knowing how to specify a path to each node is essential to setting up the data path. To learn more about path syntax, see [Reference a path to enriched nodes](#) and [skillset definition](#) for examples.

Run the indexer

Once you have created a data source, indexes, and skillset, you're ready to [create and run the indexer](#). This step puts the pipeline into execution.

You can query your search index after processing concludes to test your solution.

Content lifecycle

Depending on the underlying data source, an indexer can usually provide ongoing change tracking and deletion detection. This section explains the content lifecycle of one-to-many

indexing as it relates to data refresh.

For data sources that provide change tracking and deletion detection, an indexer process can pick up changes in your source data. Each time you run the indexer and skillset, the index projections are updated if the skillset or underlying source data has changed. Any changes picked up by the indexer are propagated through the enrichment process to the projections in the index, ensuring that your projected data is a current representation of content in the originating data source. Data refresh activity is captured in a projected key value for each chunk. This value gets updated when the underlying data changes.

 **Note**

While you can manually edit the data in the projected documents using the [index push API](#), you should avoid doing so. Manual updates to an index are overwritten on the next pipeline invocation, assuming the document in source data is updated and the data source has change tracking or deletion detection enabled.

Updated content

If you add new content to your data source, new chunks or child documents are added to the index on the next indexer run.

If you modify existing content in the data source, chunks are updated incrementally in the search index if the data source you're using supports change tracking and deletion detection. For example, if a word or sentence changes in a document, the chunk in the target index that contains that word or sentence is updated on the next indexer run. Other types of updates, such as changing a field type and some attributions, aren't supported for existing fields. For more information about allowed updates, see [Update an index schema](#).

Some data sources like [Azure Storage](#) support change and deletion tracking by default, based on the timestamp. Other data sources such as [OneLake](#), [Azure SQL](#), or [Azure Cosmos DB](#) must be configured for change tracking.

Deleted content

If the source content no longer exists (for example, if text is shortened to have fewer chunks), the corresponding child document in the search index is deleted. The remaining child documents also get their key updated to include a new hash value, even if their content didn't otherwise change.

If a parent document is completely deleted from the datasource, the corresponding child documents only get deleted if the deletion is detected by a `dataDeletionDetectionPolicy` defined on the datasource definition. If you don't have a `dataDeletionDetectionPolicy` configured and need to delete a parent document from the datasource, then you should manually delete the child documents if they're no longer wanted.

Projected key value

To ensure data integrity for updated and deleted content, data refresh in one-to-many indexing relies on a *projected key value* on the "many" side. If you're using integrated vectorization or the [Import and vectorize data wizard](#), the projected key value is the `parent_id` field in a chunked or "many" side of the index.

A projected key value is a unique identifier that the indexer generates for each document. It ensures uniqueness and allows for change and deletion tracking to work correctly. This key contains the following segments:

- A random hash to guarantee uniqueness. This hash changes if the parent document is updated on subsequent indexer runs.
- The parent document's key.
- The enrichment annotation path that identifies the context that that document was generated from.

For example, if you split a parent document with key value "aa1b22c33" into four pages, and then each of those pages is projected as its own document via index projections:

- aa1b22c33
- aa1b22c33_pages_0
- aa1b22c33_pages_1
- aa1b22c33_pages_2

If the parent document is updated in the source data, perhaps resulting in more chunked pages, the random hash changes, more pages are added, and the content of each chunk is updated to match whatever is in the source document.

Example of separate parent-child indexes

This section shows examples for separate parent and child indexes. It's an uncommon pattern, but it's possible you might have application requirements that are best met using this approach. In this scenario, you're projecting parent-child content into two separate indexes.

Each schema has the fields for its particular grain, with the parent ID field common to both indexes for use in a [lookup query](#). The primary search corpus is the child index, but then issue a lookup query to retrieve the parent fields for each match in the result. Azure AI Search doesn't support joins at query time, so your application code or orchestration layer would need to merge or collate results that can be passed to an app or process.

The parent index has a parent_id field and title. The parent_id is the document key. You don't need vector search configuration unless you want to vectorize fields at the parent document level.

JSON

```
{  
    "name": "my-parent-index",  
    "fields": [  
  
        {"name": "parent_id", "type": "Edm.String", "filterable": true},  
        {"name": "title", "type": "Edm.String", "searchable": true, "filterable":  
true, "sortable": true, "retrievable": true},  
    ]  
}
```

The child index has the chunked fields, plus the parent_id field. If you're using integrated vectorization, scoring profiles, semantic ranker, or analyzers you would set these in the child index.

JSON

```
{  
    "name": "my-child-index",  
    "fields": [  
        {"name": "chunk_id", "type": "Edm.String", "key": true, "filterable":  
true, "analyzer": "keyword"},  
        {"name": "parent_id", "type": "Edm.String", "filterable": true},  
        {"name": "chunk", "type": "Edm.String", "searchable": true, "retrievable":  
true},  
        {"name": "chunk_vector", "type": "Collection(Edm.Single)", "searchable":  
true, "retrievable": false, "stored": false, "dimensions": 1536,  
"vectorSearchProfile": "hnsw"}  
    ],  
    "vectorSearch": {  
        "algorithms": [{"name": "hnsw", "kind": "hnsw", "hnswParameters": {}}],  
        "profiles": [{"name": "hnsw", "algorithm": "hnsw"}]  
    },  
    "scoringProfiles": [],  
    "semanticConfiguration": [],  
    "analyzers": []  
}
```

Here's an example of an index projection definition that specifies the data path the indexer should use to index content. It specifies the child index name in the index projection definition, and it specifies the mappings of every child or chunk-level field. This is the only place the child index name is specified.

JSON

```
"indexProjections": {  
    "selectors": [  
        {  
            "targetIndexName": "my-child-index",  
            "parentKeyFieldName": "parent_id",  
            "sourceContext": "/document/pages/*",  
            "mappings": [  
                {  
                    "name": "chunk",  
                    "source": "/document/pages/*",  
                    "sourceContext": null,  
                    "inputs": []  
                },  
                {  
                    "name": "chunk_vector",  
                    "source": "/document/pages/*/chunk_vector",  
                    "sourceContext": null,  
                    "inputs": []  
                }  
            ]  
        }  
    ]  
}
```

The indexer definition specifies the components of the pipeline. In the indexer definition, the index name to provide is the parent index. If you need field mappings for the parent-level fields, define them in outputFieldMappings. For one-to-many indexing that uses separate indexes, the indexer definition might look like the following example.

JSON

```
{  
    "name": "my-indexer",  
    "dataSourceName": "my-ds",  
    "targetIndexName": "my-parent-index",  
    "skillsetName" : "my-skillset"  
    "parameters": { },  
    "fieldMappings": (optional) Maps fields in the underlying data source to fields  
in an index,  
    "outputFieldMappings" : (required) Maps skill outputs to fields in an index,  
}
```

Next step

Data chunking and one-to-many indexing are part of the RAG pattern in Azure AI Search. Continue on to the following tutorial and code sample to learn more about it.

[How to build a RAG solution using Azure AI Search](#)

Debug Sessions in Azure AI Search

Debug Sessions is a visual editor that works with an existing skillset in the Azure portal, exposing the structure and content of a single enriched document as it's produced by an indexer and skillset for the duration of the session. Because you're working with a live document, the session is interactive - you can identify errors, modify and invoke skill execution, and validate the results in real time. If your changes resolve the problem, you can commit them to a published skillset to apply the fixes globally.

This article explains supported scenarios and how the editor is organized. Tabs and sections of the editor unpack different layers of the skillset so that you can examine skillset structure, flow, and the content it generates at run time.

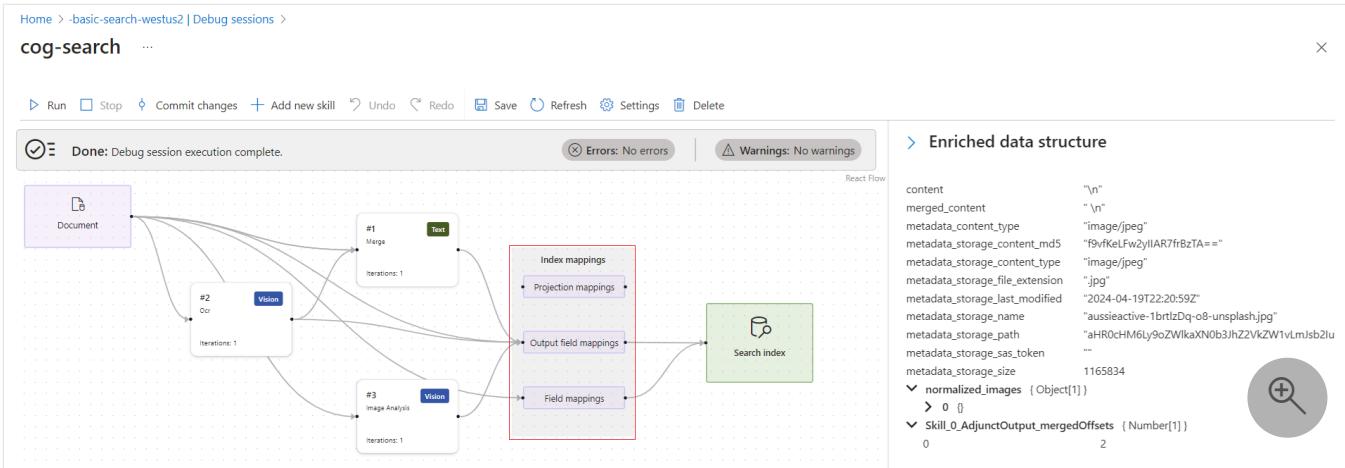
Supported scenarios

Use Debug Sessions to investigate and resolve problems with:

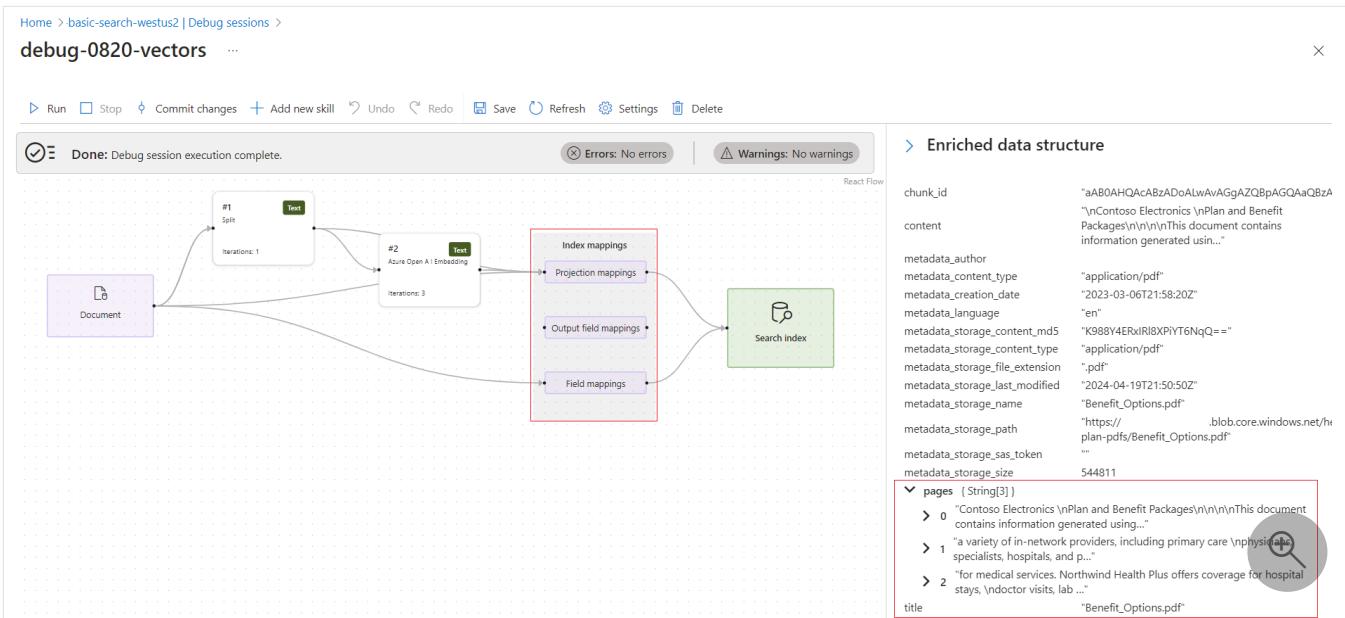
- Built-in skills used for [AI enrichment](#), such as OCR, image analysis, entity recognition, and keyword extraction.
- Built-in skills used for [integrated vectorization](#), with data chunking through Text Split, and vectorization through an embedding skill.
- Custom skills used to integrate external processing that you provide.

Compare the following debug session images for the first two scenarios. For both scenarios, the surface area shows the progression of skills that generate or transform content en route from the source document to the search index. The flow includes index mapping options, and you can trace the arrows to follow the processing trail. The details pane to the right is context-sensitive. It shows a representation of the enriched document that's created by the pipeline, or the details of a skill or mapping.

The first image shows a pattern for applied AI enrichment (no vectors). Skills can run sequentially or in parallel if there are no dependencies. Index mappings show how enriched or generated content travels from in-memory data structures to fields in an index. Enriched document shows the data structure that the skillset creates.



The second image shows a typical pattern for integrated vectorization. Skills for integrated vectorization usually include a Text Split skill and an embedding skill. A Text Split skill divides a document into chunks. An embedding skill calls an embedding API to vectorize those chunks. This particular skillset chunks content into an array of "pages". For integrated vectorization, projection mappings control how chunks are mapped to fields in the index.



Limitations

Debug Sessions work with all generally available [indexer data sources](#) and most preview data sources, with the following exceptions:

- SharePoint indexer.
- Azure Cosmos DB for MongoDB indexer.
- For the Azure Cosmos DB for NoSQL, if a row fails during index and there's no corresponding metadata, the debug session might not pick the correct row.

- For the SQL API of Azure Cosmos DB, if a partitioned collection was previously non-partitioned, the debug session won't find the document.
- For custom skills, a user-assigned managed identity isn't supported for a debug session connection to Azure Storage. As stated in the prerequisites, you can use a system managed identity, or specify a full access connection string that includes a key. For more information, see [Connect a search service to other Azure resources using a managed identity](#).
- Data sources with encryption enabled via [customer managed keys \(CMK\)](#).
- Currently, the ability to select which document to debug is unavailable. This limitation isn't permanent and should be lifted soon. At this time, Debug Sessions selects the first document in the source data container or folder.

How a debug session works

When you start a session, the search service creates a copy of the skillset, indexer, and a data source containing a single document used to test the skillset. All session state is saved to a new blob container created by the Azure AI Search service in an Azure Storage account that you provide. The name of the generated container has a prefix of `ms-az-cognitive-search-debugsession`. The prefix is required because it mitigates the chance of accidentally exporting session data to another container in your account.

A cached copy of the enriched document and skillset is loaded into the visual editor so that you can inspect the content and metadata of the enriched document, with the ability to check each document node and edit any aspect of the skillset definition. Any changes made within the session are cached. Those changes won't affect the published skillset unless you commit them. Committing changes will overwrite the production skillset.

If the enrichment pipeline doesn't have any errors, a debug session can be used to incrementally enrich a document, test and validate each change before committing the changes.

Debug sessions help identify the root cause of errors or warnings by analyzing the data, skill inputs and outputs, and field mappings. If the indexer encounters configuration issues, such as incorrect network setup, permission-related access errors, or similar, please review the specific error message along with the linked documentation provided. For troubleshooting guidance, refer to the [common indexer errors and warnings](#).

Debug Sessions with private connectivity

If your AI enrichment pipeline uses shared private links to access Azure resources, additional configuration is required to ensure indexer and debug sessions work correctly. This includes permissions, trusted access, and network setup.

- If you're using [managed identity](#), assign the necessary roles to your search service identity, including `Storage Blob Data Contributor`, so debug sessions can write session data to your storage account.
- Ensure the search service has access to all resources referenced in the [skillset definition](#), including any used in the debug session.
- In your storage account, [enable trusted services](#) to allow access from Azure AI Search.
- Set `"executionEnvironment" = "private"` property in the indexer definition to ensure the indexer runs in a private context.
- Create a [shared private link](#) for each resource accessed by the search service, including: your data source, if configured to indexer AI enrichment cache and knowledge store, and any other resources configured in your skillset.
- For other troubleshooting guidance, refer to the [common indexer errors and warnings](#).

Debug session layout

The visual editor is organized into a surface area showing a progression of operations, starting with document cracking, followed by skills, mappings, and an index.

Select any skill or mapping, and a pane opens to side showing relevant information.

Home > -basic-search-westus2 | Debug sessions >
debug-0820-vectors ...

Run Stop Commit changes Add new skill Undo Redo Save

Done: Debug session execution complete. Errors: No errors Warnings:

Skill: #2

Iterations (3) Skill Settings Errors/Warnings (0)

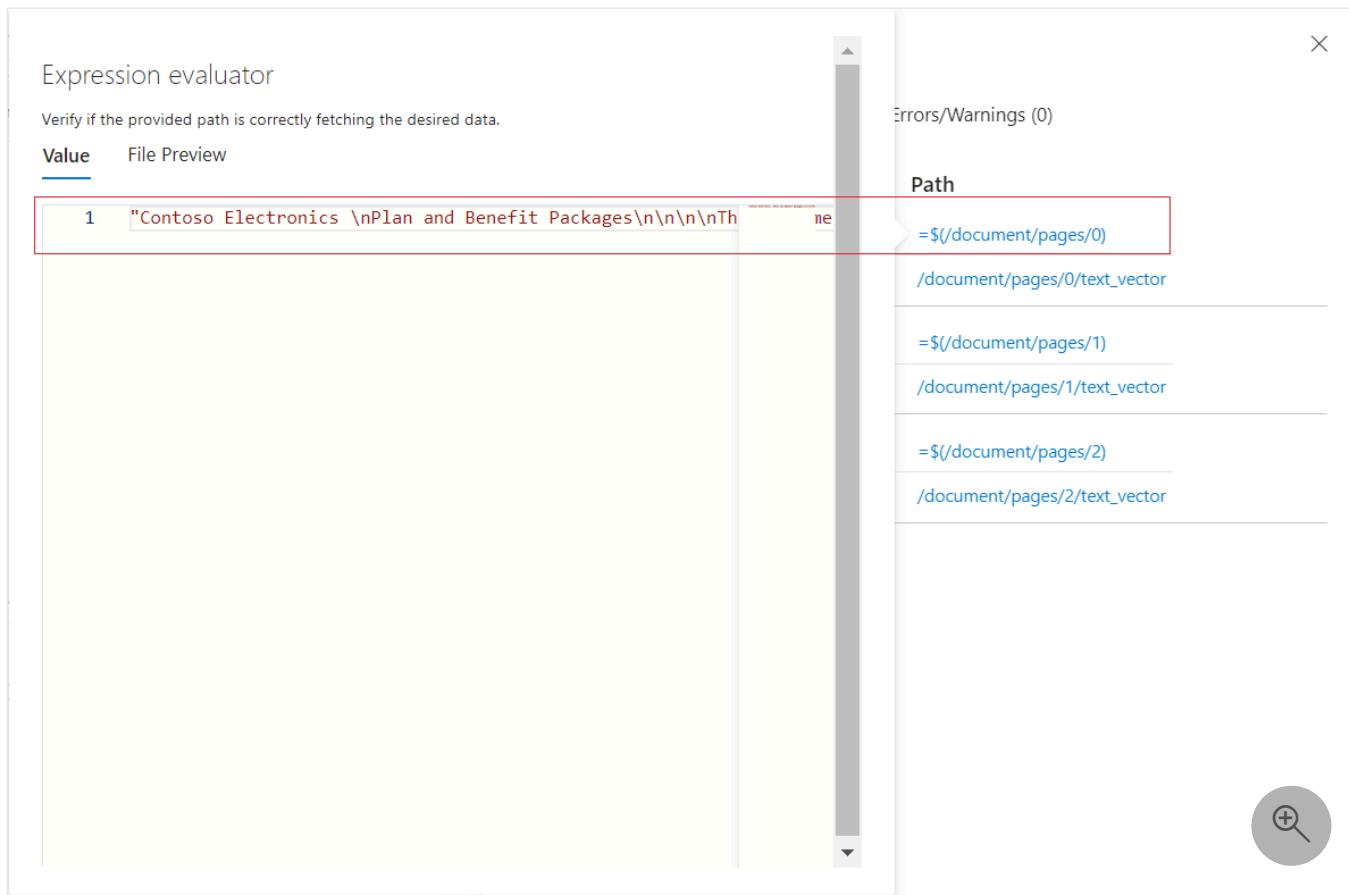
Iteration	Property	Name	Path
0	Inputs	text	=\${/document/pages/0}
	Outputs	embedding	/document/pages/0/text_vector
1	Inputs	text	=\${/document/pages/1}
	Outputs	embedding	/document/pages/1/text_vector
2	Inputs	text	=\${/document/pages/2}
	Outputs	embedding	/document/pages/2/text_vector

Document

#1 Split
Iterations: 1

#2 Azure Open AI Embedding
Iterations: 3

Follow the links to drill further into skills processing. For example, the following screenshot shows the output of the first iteration of the Text Split skill.



Skill details pane

The **Skill details** pane has the following sections:

- **Iterations:** Shows you how many times a skill executes. You can check the inputs and outputs of each one.
- **Skill Settings:** View or edit the JSON skillset definition.
- **Errors and warnings:** Shows the errors or warnings specific to this skill.

Enriched data structure pane

The **Enriched Data Structure** pane slides out to the side when you select the blue show or hide arrow symbol. It's a human readable representation of what the enriched document contains. Previous screenshots in this article show examples of the enriched data structure.

Next steps

Now that you understand the elements of debug sessions, start your first debug session on an existing skillset.

[How to debug a skillset](#)

Last updated on 10/21/2025

Debug an Azure AI Search skillset in Azure portal

05/08/2025

Start a portal-based debug session to identify and resolve errors, validate changes, and push changes to an existing skillset in your Azure AI Search service.

A debug session is a cached indexer and skillset execution, scoped to a single document, that you can use to edit and test skillset changes interactively. When you're finished debugging, you can save your changes to the skillset.

For background on how a debug session works, see [Debug sessions in Azure AI Search](#). To practice a debug workflow with a sample document, see [Tutorial: Debug sessions](#).

Prerequisites

- An Azure AI Search service, any region or tier.
- An Azure Storage account, used to save session state.
- An existing enrichment pipeline, including a data source, a skillset, an indexer, and an index.

Security and permissions

- To save a debug session to Azure storage, the search service identity must have **Storage Blob Data Contributor** permissions on Azure Storage. Otherwise, plan on choosing a full access connection string for the debug session connection to Azure Storage.
- If the Azure Storage account is behind a firewall, configure it to [allow search service access](#).

Limitations

Debug sessions work with all generally available [indexer data sources](#) and most preview data sources, with the following exceptions:

- SharePoint Online indexer.
- Azure Cosmos DB for MongoDB indexer.

- For the Azure Cosmos DB for NoSQL, if a row fails during index and there's no corresponding metadata, the debug session might not pick the correct row.
- For the SQL API of Azure Cosmos DB, if a partitioned collection was previously non-partitioned, the debug session won't find the document.
- For custom skills, a user-assigned managed identity isn't supported for a debug session connection to Azure Storage. As stated in the prerequisites, you can use a system managed identity, or specify a full access connection string that includes a key. For more information, see [Connect a search service to other Azure resources using a managed identity](#).

Create a debug session

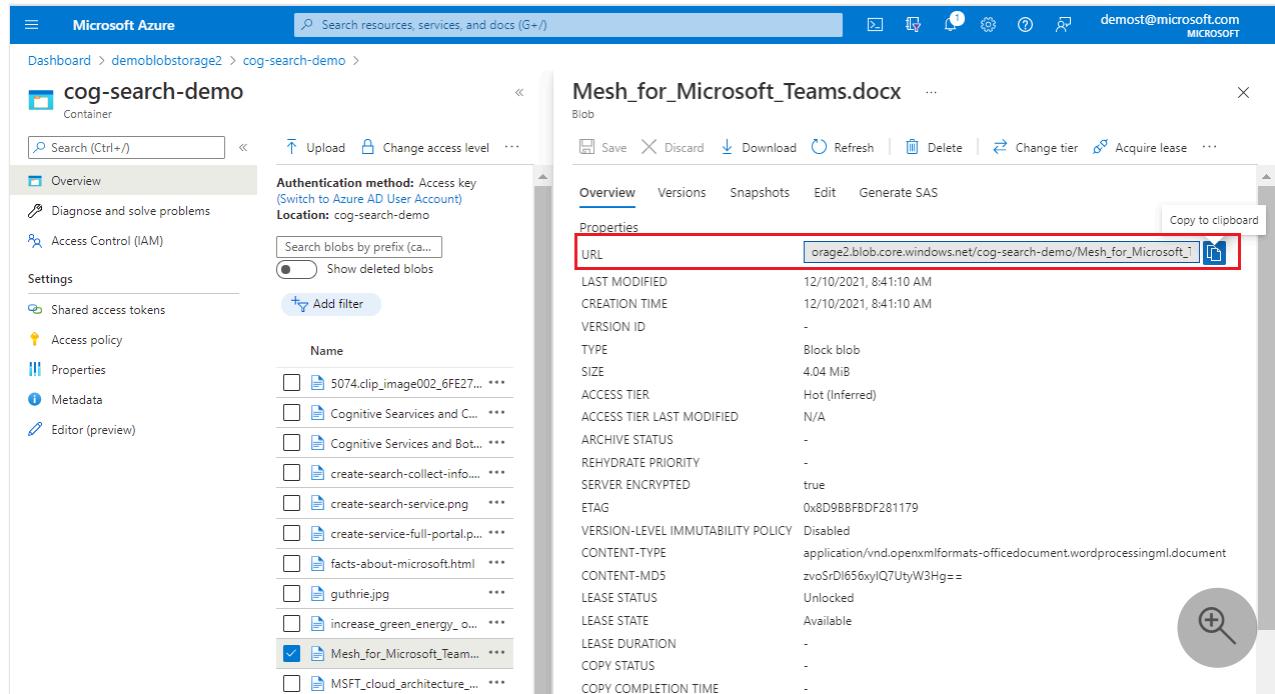
1. Sign in to the [Azure portal](#) and [find your search service](#).
2. On the left menu, select **Search management > Debug sessions**.
3. On the action bar at the top, select **Add debug session**.

The screenshot shows the Azure portal interface for managing a search service named 'demo-srch'. The main area displays a table for 'Debug sessions' with one entry: 'No debug sessions were found'. The top navigation bar includes a 'Search' bar, a 'Filter by name...' bar, and buttons for 'Add debug session', 'Refresh', and 'Delete'. The left sidebar lists various management sections: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Search management (with sub-options like Indexes, Indexers, Data sources, Aliases, Skillsets, and the currently selected 'Debug sessions'), Settings, and Semantic ranker. A large circular button with a plus sign and a magnifying glass icon is located in the bottom right corner of the main content area.

4. In **Debug session name**, provide a name that will help you remember which skillset, indexer, and data source the debug session is about.
5. In **Indexer template**, select the indexer that drives the skillset you want to debug. Copies of both the indexer and skillset are used to initialize the session.

6. In **Document to debug**, choose the first document in the index or select a specific document. If you select a specific document, depending on the data source, you're asked for a URI or a row ID.

If your specific document is a blob, provide the blob URI. You can find the URI in the blob property page in the Azure portal.



The screenshot shows the Microsoft Azure Storage Blob properties page. On the left, there's a sidebar with options like Overview, Diagnose and solve problems, Access Control (IAM), Settings, Shared access tokens, Access policy, Properties, Metadata, and Editor (preview). The main area shows a list of blobs with their names and preview icons. One blob, "Mesh_for_Microsoft_Teams.docx", is selected and highlighted with a blue border. On the right, there's a detailed properties pane with tabs for Overview, Versions, Snapshots, Edit, and Generate SAS. The "Overview" tab is selected. Under the "Properties" section, the "URL" field is highlighted with a red box and contains the value "https://cog-search-demo.cognitiveservices.azure.com/blobs/cog-search-demo/Mesh_for_Microsoft_Teams.docx?sv=2021-07-01&st=2021-10-12T08%3A41%3A10Z&se=2021-10-12T08%3A41%3A10Z&srt=b&sp=r&sr=b&sig=0wD9BBFBDF281179&api-version=2021-07-01". There are also "Save", "Discard", "Download", "Refresh", "Delete", "Change tier", "Acquire lease", and "Copy to clipboard" buttons at the top of the properties pane.

7. In **Storage account**, choose a general-purpose storage account for caching the debug session.

8. Select **Authenticate using managed identity** if you previously assigned **Storage Blob Data Contributor** permissions to the search service system-managed identity. If you don't check this box, the search service connects using a full access connection string.

9. Select **Save**.

- Azure AI Search creates a blob container on Azure Storage named *ms-az-cognitive-search-debugsession*.
- Within that container, it creates a folder using the name you provided for the session name.
- It starts your debug session.

The debug session begins by executing the indexer and skillset on the selected document. The document's content and metadata are visible and available in the session.

A debug session can be canceled while it's executing. If you hit the **Cancel** button you should be able to analyze partial results.

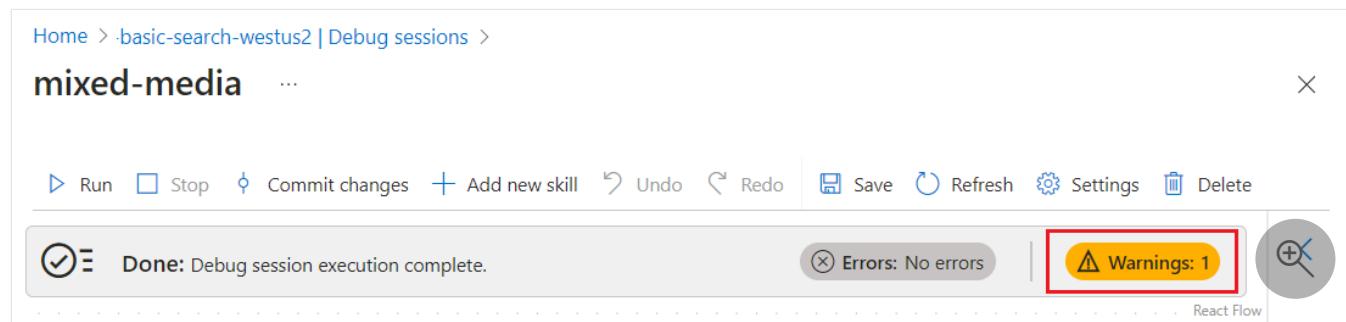
It's expected for a debug session to take longer to execute than the indexer since it goes through extra processing.

Start with errors and warnings

Indexer execution history in the Azure portal gives you the full error and warning list for all documents. In a debug session, the errors and warnings are limited to one document. You can work through this list, make your changes, and then return to the list to verify whether issues are resolved.

Remember that a debug session is based on one document from the entire index. If an input or output looks wrong, the problem could be specific to that document. You can choose a different document to confirm whether errors and warnings are pervasive or specific to a single document.

Select **Errors** or **Warnings** for a list of issues.



As a best practice, resolve problems with inputs before moving on to outputs.

To prove whether a modification resolves an error, follow these steps:

1. Select **Save** in the skill details pane to preserve your changes.
2. Select **Run** in the session window to invoke skillset execution using the modified definition.
3. Return to **Errors** or **Warnings** to see if the count is reduced.

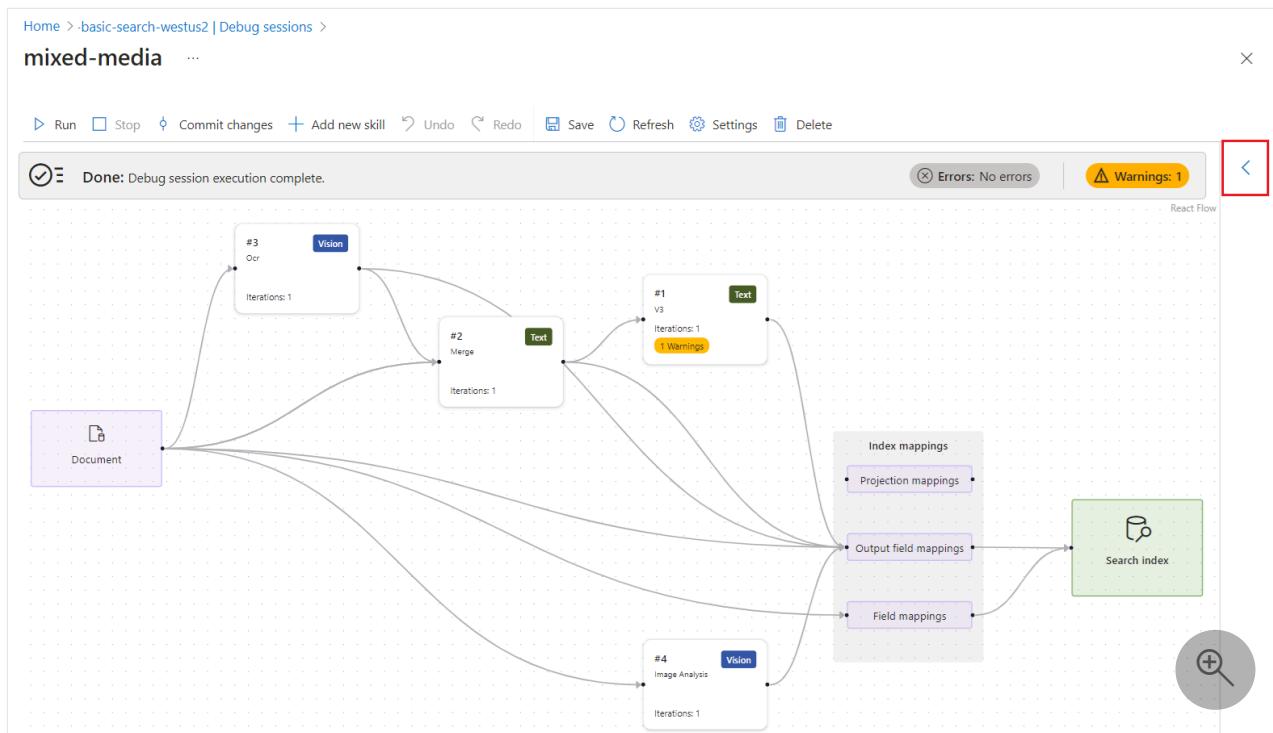
View enriched or generated content

AI enrichment pipelines extract or infer information and structure from source documents, creating an enriched document in the process. An enriched document is first created during document cracking and populated with a root node (`/document`), plus nodes for any content that is lifted directly from the data source, such as metadata and the document key. More

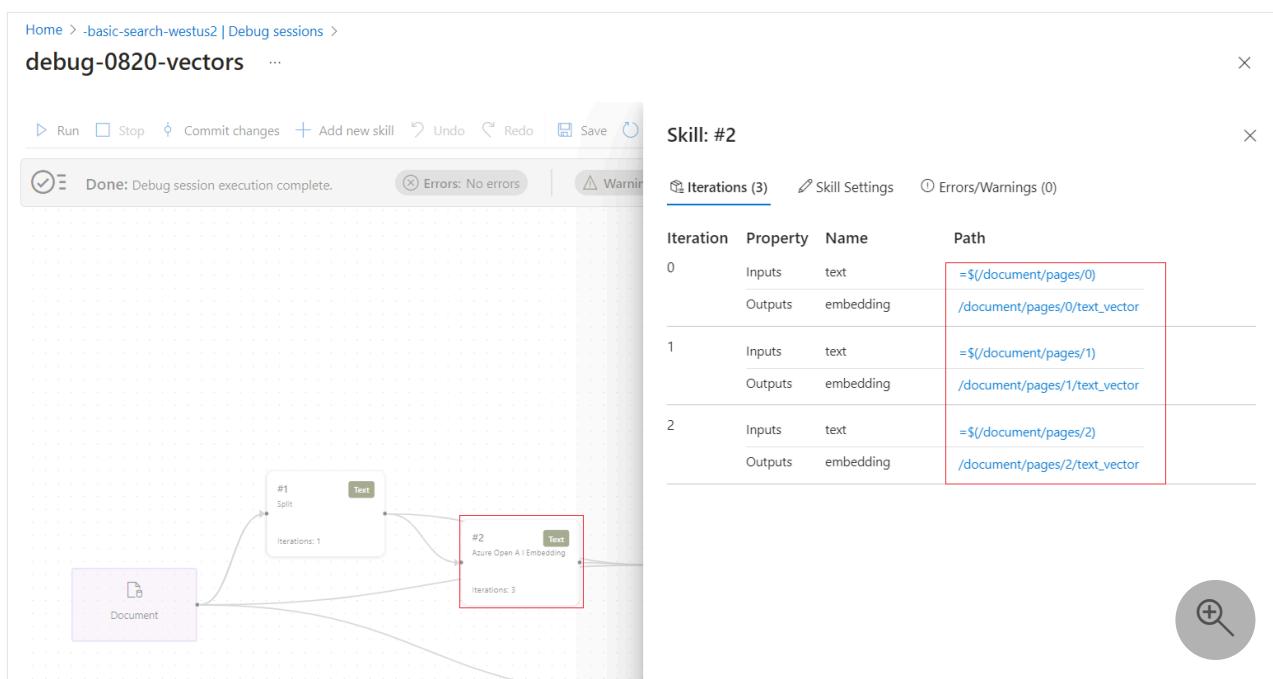
nodes are created by skills during skill execution, where each skill output adds a new node to the enrichment tree.

All content created or used by a skillset appears in the Expression Evaluator. You can hover over the links to view each input or output value in the enriched document tree. To view the input or output of each skill, follow these steps:

1. In a debug session, expand the blue arrow to view context-sensitive details. By default, the detail is the enriched document data structure. However, if you select a skill or a mapping, the detail is about that object.



2. Select a skill.



3. Follow the links to drill further into skills processing. For example, the following screenshot shows the output of the first iteration of the Text Split skill.

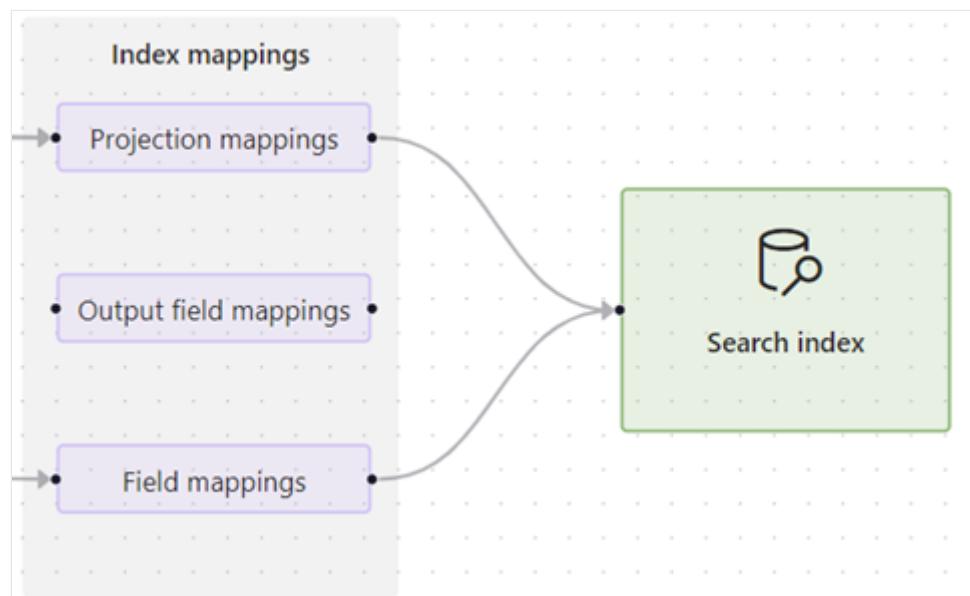
The screenshot shows the 'Expression evaluator' interface. On the left, under 'Value', the output is displayed as a red-bordered box containing the string: "Contoso Electronics \nPlan and Benefit Packages\n\n\n\n\nTh". Below this, under 'File Preview', there is a small preview icon. On the right, the 'Path' section lists several paths:

- =\${/document/pages/0}
- /document/pages/0/text_vector
- =\${/document/pages/1}
- /document/pages/1/text_vector
- =\${/document/pages/2}
- /document/pages/2/text_vector

A magnifying glass icon is located at the bottom right of the panel.

Check index mappings

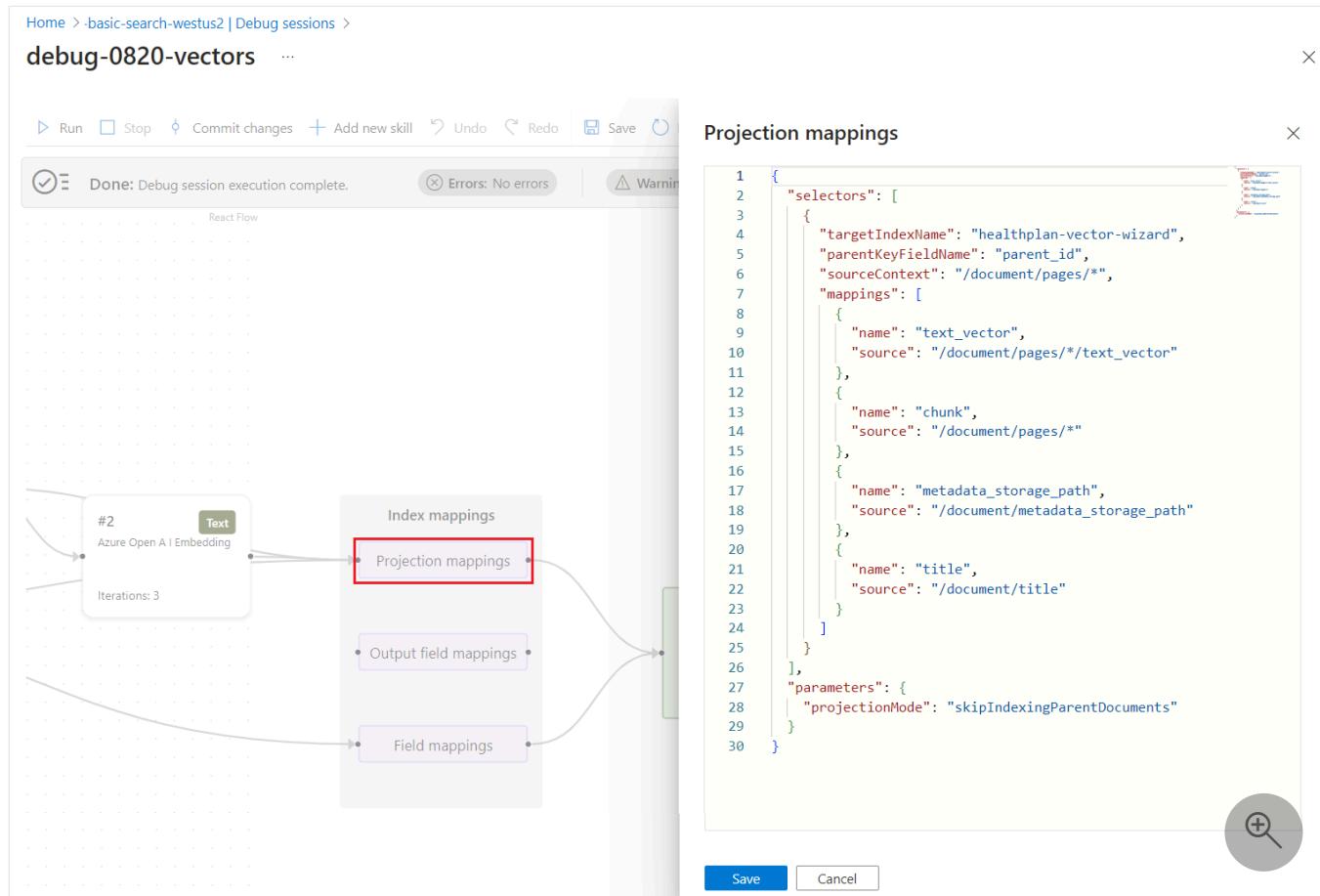
If skills produce output but the search index is empty, check the field mappings. Field mappings specify how content moves out of the pipeline and into a search index.



Select one of the mapping options and expand the details view to review source and target definitions.

- **Projection Mappings** are found in skillsets that provide integrated vectorization, such as the skills created by the [Import and vectorize data wizard](#). These mappings determine parent-child (chunk) field mappings and whether a secondary index is created for just the chunked content
- **Output Field Mappings** are found in indexers and are used when skillsets invoke built-in or custom skills. These mappings are used to set the data path from a node in the enrichment tree to a field in the search index. For more information about paths, see [enrichment node path syntax](#).
- **Field Mappings** are found in indexer definitions and they establish the data path from raw content in the data source and a field in the index. You can use field mappings to add encoding and decoding steps as well.

This example shows the details for a projection mapping. You can edit the JSON to fix any mapping issues.



The screenshot shows the Azure AI Search interface for editing a skill definition named "debug-0820-vectors". The top navigation bar includes "Home", ".basic-search-westus2 | Debug sessions", and "debug-0820-vectors". Below the navigation is a toolbar with "Run", "Stop", "Commit changes", "Add new skill", "Undo", "Redo", "Save", and a refresh icon. A message bar indicates "Done: Debug session execution complete." with "Errors: No errors" and "Warnings".

The main area displays a "React Flow" diagram. On the left, a node labeled "#2 Azure Open AI Embedding" with "Text" type and "Iterations: 3" is connected to a central "Index mappings" node. From "Index mappings", arrows point to three boxes: "Projection mappings" (which is highlighted with a red border), "Output field mappings", and "Field mappings".

To the right of the diagram is a code editor titled "Projection mappings" showing the following JSON code:

```

1  {
2    "selectors": [
3      {
4        "targetIndexName": "healthplan-vector-wizard",
5        "parentKeyFieldName": "parent_id",
6        "sourceContext": "/document/pages/*",
7        "mappings": [
8          {
9            "name": "text_vector",
10           "source": "/document/pages/*/text_vector"
11         },
12         {
13           "name": "chunk",
14           "source": "/document/pages/*"
15         },
16         {
17           "name": "metadata_storage_path",
18           "source": "/document/metadata_storage_path"
19         },
20         {
21           "name": "title",
22           "source": "/document/title"
23         }
24       ],
25     },
26   ],
27   "parameters": {
28     "projectionMode": "skipIndexingParentDocuments"
29   }
30 }

```

At the bottom of the code editor are "Save" and "Cancel" buttons, and a magnifying glass icon for search.

Edit skill definitions

If the field mappings are correct, check individual skills for configuration and content. If a skill fails to produce output, it might be missing a property or parameter, which can be determined through error and validation messages.

Other issues, such as an invalid context or input expression, can be harder to resolve because the error will tell you what is wrong, but not how to fix it. For help with context and input syntax, see [Reference enrichments in an Azure AI Search skillset](#). For help with individual messages, see [Troubleshooting common indexer errors and warnings](#).

The following steps show you how to get information about a skill.

1. Select a skill on the work surface. The Skill details pane opens to the right.
2. Edit a skill definition using **Skill Settings**. You can edit the JSON directly.
3. Check the [path syntax for referencing nodes](#) in an enrichment tree. Following are some of the most common input paths:
 - `/document/content` for chunks of text. This node is populated from the blob's content property.
 - `/document/merged_content` for chunks of text in skillets that include Text Merge skill.
 - `/document/normalized_images/*` for text that is recognized or inferred from images.

Debug a custom skill locally

Custom skills can be more challenging to debug because the code runs externally, so the debug session can't be used to debug them. This section describes how to locally debug your Custom Web API skill, debug session, Visual Studio Code and [ngrok](#) or [Tunnelmole](#). This technique works with custom skills that execute in [Azure Functions](#) or any other Web Framework that runs locally (for example, [FastAPI](#)).

Get a public URL

This section describes two approaches for getting a public URL to a custom skill.

Use Tunnelmole

Tunnelmole is an open source tunneling tool that can create a public URL that forwards requests to your local machine through a tunnel.

1. Install Tunnelmole:

- npm: `npm install -g tunnelmole`
- Linux: `curl -s https://tunnelmole.com/sh/install-linux.sh | sudo bash`
- Mac: `curl -s https://tunnelmole.com/sh/install-mac.sh --output install-mac.sh`
`&& sudo bash install-mac.sh`

- Windows: Install by using npm. Or if you don't have Node.js installed, download the [precompiled .exe file for Windows](#) and put it somewhere in your PATH.

2. Run this command to create a new tunnel:

```
Console
```

```
tmole 7071
```

You should see a response that looks like this:

```
Console
```

```
http://m5hdpb-ip-49-183-170-144.tunnelmole.net is forwarding to  
localhost:7071  
https://m5hdpb-ip-49-183-170-144.tunnelmole.net is forwarding to  
localhost:7071
```

In the preceding example, `https://m5hdpb-ip-49-183-170-144.tunnelmole.net` forwards to port `7071` on your local machine, which is the default port where Azure functions are exposed.

Use ngrok

[ngrok](#) is a popular, closed source, cross-platform application that can create a tunneling or forwarding URL, so that internet requests reach your local machine. Use ngrok to forward requests from an enrichment pipeline in your search service to your machine to allow local debugging.

1. Install ngrok.
2. Open a terminal and go to the folder with the ngrok executable.
3. Run ngrok with the following command to create a new tunnel:

```
Console
```

```
ngrok http 7071
```

ⓘ Note

By default, Azure functions are exposed on 7071. Other tools and configurations might require that you provide a different port.

4. When ngrok starts, copy and save the public forwarding URL for the next step. The forwarding URL is randomly generated.

Session Status	online												
Account	[REDACTED]												
Version	2.3.40												
Region	United States (us)												
Web Interface	http://127.0.0.1:4040												
Forwarding	http://3b7e-2a0e-41b-8f8f-0-90c8-fd24-9bd7-22b6.ngrok.io -> http://localhost:7071												
Forwarding	https://3b7e-2a0e-41b-8f8f-0-90c8-fd24-9bd7-22b6.ngrok.io -> http://localhost:7071												
Connections	<table><thead><tr><th>ttl</th><th>opn</th><th>rt1</th><th>rt5</th><th>p50</th><th>p90</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0.00</td></tr></tbody></table>	ttl	opn	rt1	rt5	p50	p90	0	0	0.00	0.00	0.00	0.00
ttl	opn	rt1	rt5	p50	p90								
0	0	0.00	0.00	0.00	0.00								

Configure in Azure portal

Once you have a public URL for your custom skill, modify your Custom Web API Skill URI within a debug session to call the Tunnelmole or ngrok forwarding URL. Be sure to append "/api/FunctionName" when using Azure Function for executing the skillset code.

You can edit the skill definition in the **Skill settings** section of the **Skill details** pane.

Test your code

At this point, new requests from your debug session should now be sent to your local Azure Function. You can use breakpoints in your Visual Studio Code to debug your code or run step by step.

Next steps

Now that you understand the layout and capabilities of the Debug Sessions visual editor, try the tutorial for a hands-on experience.

[Tutorial: Explore Debug sessions](#)

Reference a path to enriched nodes using context and source properties an Azure AI Search skillset

05/27/2025

During skillset execution, the engine builds an in-memory [enrichment tree](#) that captures each enrichment, such as recognized entities or translated text. In this article, learn how to reference an enrichment node in the enrichment tree so that you can pass output to downstream skills or specify an output field mapping for a search index field.

This article uses examples to illustrate various scenarios. For the full syntax, see [Skill context and input annotation language](#).

Background concepts

Before reviewing the syntax, let's revisit a few important concepts to better understand the examples provided later in this article.

 [Expand table](#)

Term	Description
"enriched document"	An enriched document is an in-memory structure that collects skill output as it's created and it holds all enrichments related to a document. Think of an enriched document as a tree. Generally, the tree starts at the root document level, and each new enrichment is created from a previous node as its child.
"node"	Within an enriched document, a node (sometimes referred to as an "annotation") is specific output such as the "text" or "layoutText" of the OCR skill, or an original source field value such as the content of a product ID field, or metadata copied from the source such as <code>metadata_storage_path</code> from blobs in Azure Storage.
"context"	The scope of enrichment, which is either the entire document, a portion of a document (pages or sentences), or if you're working with images, the extracted images from a document. By default, the enrichment context is at the <code>"/document"</code> level, scoped to individual documents contained in the data source. When a skill runs, the outputs of that skill become properties of the defined context .

Paths for different scenarios

Paths are specified in the "context" and "source" properties of a skillset, and in the [output field mappings](#) in an indexer.

```
{  
    "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",  
    "name": "#2",  
    "description": null,  
    "context": "/document/HotelId",  
    "defaultFromLanguageCode": null,  
    "defaultToLanguageCode": "es",  
    "suggestedFrom": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/Description"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "translatedText",  
            "targetName": "translated_text"  
        }  
    ]  
}
```

The example in the screenshot illustrates the path for an item in an Azure Cosmos DB collection.

- `context` path is `/document/HotelId` because the collection is partitioned into documents by the `/HotelId` field.
- `source` path is `/document/Description` because the skill is a translation skill, and the field that you want to translate is the `Description` field in each document.

All paths start with `/document`. An enriched document is created in the "document cracking" stage of indexer execution, when the indexer opens a document or reads in a row from the data source. Initially, the only node in an enriched document is the [root node \(`/document`\)](#), and it's the node from which all other enrichments occur.

The following list includes several common examples:

- `/document` is the root node and indicates an entire blob in Azure Storage, or a row in a SQL table.
- `/document/{key}` is the syntax for a document or item in an Azure Cosmos DB collection, where `{key}` is the actual key, such as `/document/HotelId` in the previous example.
- `/document/content` specifies the "content" property of a JSON blob.

- `/document/{field}` is the syntax for an operation performed on a specific field, such as translating the `/document/Description` field, seen in the previous example.
- `/document/pages/*` or `/document/sentences/*` become the context if you're breaking a large document into smaller chunks for processing. If "context" is `/document/pages/*`, the skill executes once over each page in the document. Because there might be more than one page or sentence, you can append `/*` to catch them all.
- `/document/normalized_images/*` is created during document cracking if the document contains images. All paths to images start with normalized_images. Since there are often multiple images embedded in a document, append `/*`.

Examples in the remainder of this article are based on the "content" field generated automatically by [Azure blob indexers](#) as part of the [document cracking](#) phase. When referring to documents from a Blob container, use a format such as `"/document/content"`, where the "content" field is part of the "document".

Example 1: Simple annotation reference

In Azure Blob Storage, suppose you have various files containing references to people's names that you want to extract using entity recognition. In the following skill definition, `"/document/content"` is the textual representation of the entire document, and "people" is an extraction of full names for entities identified as persons.

Because the default context is `"/document"`, the list of people can now be referenced as `"/document/people"`. In this specific case `"/document/people"` is an annotation, which could now be mapped to a field in an index, or used in another skill in the same skillset.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "categories": [ "Person" ],
  "defaultLanguageCode": "en",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    }
  ]
}
```

```
]  
}
```

Example 2: Reference an array within a document

This example builds on the previous one, showing you how to invoke an enrichment step multiple times over the same document. Assume the previous example generated an array of strings with 10 people names from a single document. A reasonable next step might be a second enrichment that extracts the last name from a full name. Because there are 10 names, you want this step to be called 10 times in this document, once for each person.

To invoke the right number of iterations, set the context as `"/document/people/*"`, where the asterisk (`"*"`) represents all the nodes in the enriched document as descendants of `"/document/people"`. Although this skill is only defined once in the skills array, it's called for each member within the document until all members are processed.

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
    "description": "Fictitious skill that gets the last name from a full name",  
    "uri": "http://names.azurewebsites.net/api/GetLastName",  
    "context" : "/document/people/*",  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "fullname",  
            "source": "/document/people/*"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "lastname",  
            "targetName": "last"  
        }  
    ]  
}
```

When annotations are arrays or collections of strings, you might want to target specific members rather than the array as a whole. The previous example generates an annotation called `"last"` under each node represented by the context. If you want to refer to this family of annotations, you could use the syntax `"/document/people/*/last"`. If you want to refer to a particular annotation, you could use an explicit index: `"/document/people/1/last"` to reference the last name of the first person identified in the document. Notice that in this syntax arrays are "0 indexed".

Example 3: Reference members within an array

Sometimes you need to group all annotations of a particular type to pass them to a particular skill. Consider a hypothetical custom skill that identifies the most common last name from all the last names extracted in Example 2. To provide just the last names to the custom skill, specify the context as `"/document"` and the input as `"/document/people/*/lastname"`.

Notice that the cardinality of `"/document/people/*/lastname"` is larger than that of document. There might be 10 lastname nodes while there's only one document node for this document. In that case, the system will automatically create an array of `"/document/people/*/lastname"` containing all of the elements in the document.

```
JSON

{
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
    "description": "Fictitious skill that gets the most common string from an
array of strings",
    "uri": "http://names.azurewebsites.net/api/MostCommonString",
    "context" : "/document",
    "inputs": [
        {
            "name": "strings",
            "source": "/document/people/*/lastname"
        }
    ],
    "outputs": [
        {
            "name": "mostcommon",
            "targetName": "common-lastname"
        }
    ]
}
```

Tips for annotation path troubleshooting

If you're having trouble with specifying skill inputs, these tips might help you move forward:

- [Run the Import data wizard](#) over your data to review the skillset definitions and field mappings that the wizard generates.
- [Start a debug session](#) on a skillset to view the structure of an enriched document. You can edit the paths and other parts of the skill definition, and then run the skill to validate your changes.

See also

- [Skill context and input annotation language](#)
- [How to integrate a custom skill into an enrichment pipeline](#)
- [How to define a skillset](#)
- [Create Skillset \(REST\)](#)
- [How to map enriched fields to an index](#)

Map enriched output to fields in a search index in Azure AI Search

Article • 04/14/2025



This article explains how to set up *output field mappings*, defining a data path between in-memory data generated during [skillset processing](#), and target fields in a search index. During indexer execution, skills-generated information exists in memory only. To persist this information in a search index, you need to tell the indexer where to send the data.

An output field mapping is defined in an [indexer](#) and has the following elements:

JSON

```
"outputFieldMappings": [
  {
    "sourceFieldName": "document/path-to-a-node-in-an-enriched-document",
    "targetFieldName": "some-search-field-in-an-index",
    "mappingFunction": null
  }
],
```

In contrast with a [fieldMappings](#) definition that maps a path between verbatim source fields and index fields, an [outputFieldMappings](#) definition maps in-memory enrichments to fields in a search index.

Prerequisites

- Indexer, index, data source, and skillset.
- Index fields must be simple or top-level fields. You can't output to a [complex type](#). However, if you have a complex type, you can use an output field definition to flatten parts of the complex type and send them to a collection in a search index.

When to use an output field mapping

Output field mappings are required if your indexer has an attached [skillset](#) that creates new information that you want in your index. Examples include:

- Vectors from embedding skills
- Optical character recognition (OCR) text from image skills
- Locations, organizations, or people from entity recognition skills

Output field mappings can also be used to:

- Create multiple copies of your generated content (one-to-many output field mappings).
- Flatten a source document's complex type. For example, assume source documents have a complex type, such as a multipart address, and you want just the city. You can use an output field mapping to [flatten a nested data structure](#), and then use an output field mapping to send the output to a string collection in your search index.

Output field mappings apply to search indexes only. If you're populating a [knowledge store](#), use [projections](#) for data path configuration.

Define an output field mapping

Output field mappings are added to the `outputFieldMappings` array in an indexer definition, typically placed after the `fieldMappings` array. An output field mapping consists of three parts.

You can use the REST API or an Azure SDK to define output field mappings.

Tip

Indexers created by the [Import data wizard](#) include output field mappings generated by the wizard. If you need examples, run the wizard over your data source to see the output field mappings in the indexer.

REST APIs

1. Use [Create Indexer](#) or [Create or Update Indexer](#) or an equivalent method in an Azure SDK. Here's an example of an indexer definition.

JSON

```
{  
  "name": "myindexer",  
  "description": null,  
  "dataSourceName": "mydatasource",  
  "targetIndexName": "myindex",  
  "schedule": { },  
  "parameters": { },
```

```

    "fieldMappings": [],
    "outputFieldMappings": [],
    "disabled": false,
    "encryptionKey": { }
}

```

2. Fill out the `outputFieldMappings` array to specify the mappings. A field mapping consists of three parts.

JSON

```

"outputFieldMappings": [
{
  "sourceFieldName": "/document/path-to-a-node-in-an-enriched-
document",
  "targetFieldName": "some-search-field-in-an-index",
  "mappingFunction": null
}
]

```

[\[\] Expand table](#)

Property	Description
sourceFieldName	Required. Specifies a path to enriched content. An example might be <code>/document/content</code> . See Reference enrichments in an Azure AI Search skillset for path syntax and examples.
targetFieldName	Optional. Specifies the search field that receives the enriched content. Target fields must be top-level simple fields or collections. It can't be a path to a subfield in a complex type. If you want to retrieve specific nodes in a complex structure, you can flatten individual nodes in memory, and then send the output to a string collection in your index.
mappingFunction	Optional. Adds extra processing provided by mapping functions supported by indexers. For enrichment nodes, encoding and decoding are the most commonly used functions.

3. The `targetFieldName` is always the name of the field in the search index.
4. The `sourceFieldName` is a path to a node in the enriched document. It's the output of a skill. The path always starts with `/document`, and if you're indexing from a blob, the second element of the path is `/content`. The third element is the value produced by the skill. For more information and examples, see [Reference enrichments in an Azure AI Search skillset](#).

This example adds entities and sentiment labels extracted from a blob's content property to fields in a search index.

JSON

```
{  
    "name": "myIndexer",  
    "dataSourceName": "myDataSource",  
    "targetIndexName": "myIndex",  
    "skillsetName": "myFirstSkillSet",  
    "fieldMappings": [],  
    "outputFieldMappings": [  
        {  
            "sourceFieldName":  
                "/document/content/organizations/*/description",  
            "targetFieldName": "descriptions",  
            "mappingFunction": {  
                "name": "base64Decode"  
            }  
        },  
        {  
            "sourceFieldName": "/document/content/organizations",  
            "targetFieldName": "orgNames"  
        },  
        {  
            "sourceFieldName": "/document/content/sentiment",  
            "targetFieldName": "sentiment"  
        }  
    ]  
}
```

5. Assign any [mapping functions](#) needed to transform the content of a field before it's stored in the index. For enrichment nodes, encoding and decoding are the most commonly used functions.

One-to-many output field mapping

You can use an output field mapping to route a single source field to multiple fields in a search index. You might do this for comparison testing or if you want fields with different attributes.

Assume a skillset that generates embeddings for a vector field, and an index that has multiple vector fields that vary by algorithm and compression settings. Within the indexer, map the embedding skill's output to each of the multiple vector fields in a search index.

JSON

```

"outputFieldMappings": [
    { "sourceFieldName" : "/document/content/text_vector", "targetFieldName" :
"vector_hnsw" },
    { "sourceFieldName" : "/document/content/text_vector", "targetFieldName" :
"vector_eknn" },
    { "sourceFieldName" : "/document/content/text_vector", "targetFieldName" :
"vector_narrow" },
    { "sourceFieldName" : "/document/content/text_vector", "targetFieldName" :
"vector_no_stored" },
    { "sourceFieldName" : "/document/content/text_vector", "targetFieldName" :
"vector_scalar" }
]

```

The source field path is skill output. In this example, the output is *text_vector*. Target name is an optional property. If you don't give the output mapping a target name, the path would be *embedding* or more precisely, */document/content/embedding*.

JSON

```
{
    "name": "test-vector-size-ss",
    "description": "Generate embeddings using Azure OpenAI Service",
    "skills": [
        {
            "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
            "name": "#1",
            "description": null,
            "context": "/document/content",
            "resourceUri": "https://my-demo-eastus.openai.azure.com",
            "apiKey": null,
            "deploymentId": "text-embedding-ada-002",
            "dimensions": 1536,
            "modelName": "text-embedding-ada-002",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/content"
                }
            ],
            "outputs": [
                {
                    "name": "embedding",
                    "targetName": "text_vector"
                }
            ],
            "authIdentity": null
        }
    ]
}
```

Flatten complex structures into a string collection

If your source data is composed of nested or hierarchical JSON, you can't use field mappings to set up the data paths. Instead, your search index must mirror the source data structure for at each level for a full import.

This section walks you through an import process that produces a one-to-one reflection of a complex document on both the source and target sides. Next, it uses the same source document to illustrate the retrieval and flattening of individual nodes into string collections.

Here's an example of a document in Azure Cosmos DB with nested JSON:

```
JSON

{
  "palette": "primary colors",
  "colors": [
    {
      "name": "blue",
      "medium": [
        "acrylic",
        "oil",
        "pastel"
      ]
    },
    {
      "name": "red",
      "medium": [
        "acrylic",
        "pastel",
        "watercolor"
      ]
    },
    {
      "name": "yellow",
      "medium": [
        "acrylic",
        "watercolor"
      ]
    }
  ]
}
```

If you wanted to fully index this source document, you'd create an index definition where the field names, levels, and types are reflected as a complex type. Because field mappings aren't supported for complex types in the search index, your index definition must mirror the source document.

```
JSON
```

```
{
  "name": "my-test-index",
  "defaultScoringProfile": "",
  "fields": [
    { "name": "id", "type": "Edm.String", "searchable": false, "retrievable": true, "key": true},
    { "name": "palette", "type": "Edm.String", "searchable": true, "retrievable": true },
    { "name": "colors", "type": "Collection(Edm.ComplexType)", "fields": [
      {
        "name": "name",
        "type": "Edm.String",
        "searchable": true,
        "retrievable": true
      },
      {
        "name": "medium",
        "type": "Collection(Edm.String)",
        "searchable": true,
        "retrievable": true,
      }
    ]
  }
}
```

Here's a sample indexer definition that executes the import. Notice there are no field mappings and no skillset.

JSON

```
{
  "name": "my-test-indexer",
  "dataSourceName": "my-test-ds",
  "skillsetName": null,
  "targetIndexName": "my-test-index",

  "fieldMappings": [],
  "outputFieldMappings": []
}
```

The result is the following sample search document, similar to the original in Azure Cosmos DB.

JSON

```
{
  "value": [
    {
      "@search.score": 1,
```

```

"id": "11bb11bb-cc22-dd33-ee44-55ff55ff55ff",
"palette": "primary colors",
"colors": [
  {
    "name": "blue",
    "medium": [
      "acrylic",
      "oil",
      "pastel"
    ]
  },
  {
    "name": "red",
    "medium": [
      "acrylic",
      "pastel",
      "watercolor"
    ]
  },
  {
    "name": "yellow",
    "medium": [
      "acrylic",
      "watercolor"
    ]
  }
]
}

```

An alternative rendering in a search index is to flatten individual nodes in the source's nested structure into a string collection in a search index.

To accomplish this task, you'll need an `outputFieldMappings` that maps an in-memory node to a string collection in the index. Although output field mappings primarily apply to skill outputs, you can also use them to address nodes after [document cracking](#) where the indexer opens a source document and reads it into memory.

The following sample index definition uses string collections to receive flattened output:

JSON

```
{
  "name": "my-new-flattened-index",
  "defaultScoringProfile": "",
  "fields": [
    { "name": "id", "type": "Edm.String", "searchable": false, "retrievable": true, "key": true },
    { "name": "palette", "type": "Edm.String", "searchable": true, "retrievable": true },
    { "name": "blue", "type": "Edm.String", "searchable": false, "retrievable": true },
    { "name": "red", "type": "Edm.String", "searchable": false, "retrievable": true },
    { "name": "yellow", "type": "Edm.String", "searchable": false, "retrievable": true }
  ]
}
```

```

        { "name": "color_names", "type": "Collection(Edm.String)", "searchable": true,
      "retrievable": true },
        { "name": "color_mediums", "type": "Collection(Edm.String)", "searchable": true,
      "retrievable": true}
    ]
}

```

Here's the sample indexer definition, using `outputFieldMappings` to associate the nested JSON with the string collection fields. Notice that the source field uses the path syntax for enrichment nodes, even though there's no skillset. Enriched documents are created in the system during document cracking, which means you can access nodes in each document tree as long as those nodes exist when the document is cracked.

JSON

```
{
  "name": "my-test-indexer",
  "dataSourceName": "my-test-ds",
  "skillsetName": null,
  "targetIndexName": "my-new-flattened-index",
  "parameters": { },
  "fieldMappings": [ ],
  "outputFieldMappings": [
    {
      "sourceFieldName": "/document/colors/*/name",
      "targetFieldName": "color_names"
    },
    {
      "sourceFieldName": "/document/colors/*/medium",
      "targetFieldName": "color_mediums"
    }
  ]
}
```

Results from the definition are as follows. Simplifying the structure loses context in this case. There's no longer any associations between a given color and the mediums it's available in. However, depending on your scenario, a result similar to the following example might be exactly what you need.

JSON

```
{
  "value": [
    {
      "@search.score": 1,
      "id": "11bb11bb-cc22-dd33-ee44-55ff55ff55ff",
      "palette": "primary colors",
      "color_names": [
        "blue",
        "red",
        "green"
      ]
    }
  ]
}
```

```
        "red",
        "yellow"
    ],
    "color_mediums": [
        "[\"acrylic\", \"oil\", \"pastel\"]",
        "[\"acrylic\", \"pastel\", \"watercolor\"]",
        "[\"acrylic\", \"watercolor\"]"
    ]
}
```

Related content

- [Field mappings and transformations](#)
- [AI enrichment overview](#)
- [Skillset concepts](#)

Extract text and information from images by using AI enrichment

Images often contain useful information that's relevant in search scenarios. Azure AI Search doesn't query image content in real time, but you can extract information about an image during indexing and make that content searchable. To represent images in a search index, you can use these approaches:

- [Vectorize images](#) to represent visual content as a searchable vector.
- [Verbalize images](#) using the GenAI Prompt skill that sends a verbalization request to a chat completion model to describe the image.
- [Analyze images](#) using an image analysis skill to generate a text representation of an image, such as *dandelion* for a photo of a dandelion, or the color *yellow*. You can also extract metadata about the image, such as its size.
- [Use OCR](#) to extract text from photos or pictures, such as the word *STOP* in a stop sign.

You can also create a [custom skill](#) to invoke any external image processing that you want to provide.

This article focuses on image analysis and OCR, custom skills that provide external processing, working with embedded images, and overlaying visualizations on original images. If verbalization or vectorization is your preferred approach, see [Multimodal search](#) instead.

To work with image content in a skillset, you need:

- ✓ Source files that include images
- ✓ A search indexer, configured for image actions
- ✓ A skillset with built-in or custom skills that invoke OCR or image analysis
- ✓ A search index with fields to receive the analyzed text output, plus output field mappings in the indexer that establish association

Optionally, you can define projections to accept image-analyzed output into a [knowledge store](#) for data mining scenarios.

Set up source files

Image processing is indexer-driven, which means that the raw inputs must be in a [supported data source](#).

- Image analysis supports JPEG, PNG, GIF, and BMP
- OCR supports JPEG, PNG, BMP, and TIF

Images are either standalone binary files or embedded in documents, such as PDF, RTF, or Microsoft application files. A maximum of 1,000 images can be extracted from a given document. If there are more than 1,000 images in a document, the first 1,000 are extracted and then a warning is generated.

Azure Blob Storage is the most frequently used storage for image processing in Azure AI Search. There are three main tasks related to retrieving images from a blob container:

- Enable access to content in the container. If you're using a full access connection string that includes a key, the key gives you permission to the content. Alternatively, you can authenticate using a [managed identity](#) or [connect as a trusted service](#).
- [Create a data source](#) of type *azureblob* that connects to the blob container storing your files.
- Review [service tier limits](#) to make sure that your source data is under maximum size and quantity limits for indexers and enrichment.

Configure indexers for image processing

After the source files are set up, enable image normalization by setting the `imageAction` parameter in indexer configuration. Image normalization helps make images more uniform for downstream processing. Image normalization includes the following operations:

- Large images are resized to a maximum height and width to make them uniform.
- For images that have metadata that specifies orientation, image rotation is adjusted for vertical loading.

Note that enabling `imageAction` (setting this parameter to other than `none`) will incur an additional charge for image extraction according to [Azure AI Search pricing](#).

Metadata adjustments are captured in a complex type created for each image. You can't opt out of the image normalization requirement. Skills that iterate over images, such as OCR and image analysis, expect normalized images.

1. [Create or update an indexer](#) to set the configuration properties:

JSON

```
{  
  "parameters":  
  {  
    "configuration":  
    {  
      "dataToExtract": "contentAndMetadata",  
    }  
  }  
}
```

```
        "parsingMode": "default",
        "imageAction": "generateNormalizedImages"
    }
}
```

2. Set `dataToExtract` to `contentAndMetadata` (required).

3. Verify that the `parsingMode` is set to *default* (required).

This parameter determines the granularity of search documents created in the index. The default mode sets up a one-to-one correspondence so that one blob results in one search document. If documents are large, or if skills require smaller chunks of text, you can add the Text Split skill that subdivides a document into paging for processing purposes. But for search scenarios, one blob per document is required if enrichment includes image processing.

4. Set `imageAction` to enable the `normalized_images` node in an enrichment tree (required):

- `generateNormalizedImages` to generate an array of normalized images as part of document cracking.
- `generateNormalizedImagePerPage` (applies to PDF only) to generate an array of normalized images where each page in the PDF is rendered to one output image. For non-PDF files, the behavior of this parameter is similar as if you had set `generateNormalizedImages`. However, setting `generateNormalizedImagePerPage` can make indexing operation less performant by design (especially for large documents) since several images would have to be generated.

5. Optionally, adjust the width or height of the generated normalized images:

- `normalizedImageMaxWidth` in pixels. Default is 2,000. Maximum value is 10,000.
- `normalizedImageMaxHeight` in pixels. Default is 2,000. Maximum value is 10,000.

The default of 2,000 pixels for the normalized images maximum width and height is based on the maximum sizes supported by the [OCR skill](#) and the [image analysis skill](#). The [OCR skill](#) supports a maximum width and height of 4,200 for non-English languages, and 10,000 for English. If you increase the maximum limits, processing could fail on larger images depending on your skillset definition and the language of the documents.

6. Optionally, [set file type criteria](#) if the workload targets a specific file type. Blob indexer configuration includes file inclusion and exclusion settings. You can filter out files you don't want.

JSON

```
{  
  "parameters" : {  
    "configuration" : {  
      "indexedFileNameExtensions" : ".pdf, .docx",  
      "excludedFileNameExtensions" : ".png, .jpeg"  
    }  
  }  
}
```

About normalized images

When `imageAction` is set to a value other than `none`, the new `normalized_images` field contains an array of images. Each image is a complex type that has the following members:

[] [Expand table](#)

Image member	Description
data	BASE64 encoded string of the normalized image in JPEG format.
width	Width of the normalized image in pixels.
height	Height of the normalized image in pixels.
originalWidth	The original width of the image before normalization.
originalHeight	The original height of the image before normalization.
rotationFromOriginal	Counter-clockwise rotation in degrees that occurred to create the normalized image. A value between 0 degrees and 360 degrees. This step reads the metadata from the image that is generated by a camera or scanner. Usually a multiple of 90 degrees.
contentOffset	The character offset within the content field where the image was extracted from. This field is only applicable for files with embedded images. The <code>contentOffset</code> for images extracted from PDF documents is always at the end of the text on the page it was extracted from in the document. This means images appear after all text on that page, regardless of the original location of the image in the page.
pageNumber	If the image was extracted or rendered from a PDF, this field contains the page number in the PDF it was extracted or rendered from, starting from 1. If the image isn't from a PDF, this field is 0.
boundingPolygon	If the image was extracted or rendered from a PDF, this field contains the coordinates of the bounding polygon that encloses the image on the page. The polygon is represented as a nested array of points, where each point has x and y

Image member	Description
	coordinates normalized to the dimensions of the page. This only applies to images extracted using <code>imageAction: generateNormalizedImages</code> .

Sample value of `normalized_images`:

JSON

```
[  
  {  
    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",  
    "width": 500,  
    "height": 300,  
    "originalWidth": 5000,  
    "originalHeight": 3000,  
    "rotationFromOriginal": 90,  
    "contentOffset": 500,  
    "pageNumber": 2,  
    "boundingPolygon": "[[{"x":0.0,"y":0.0},{x:500.0,y:0.0},  
    {"x":0.0,y:300.0},{x:500.0,y:300.0}]]"  
  }  
]
```

(!) Note

Bounding polygon data is represented as a string containing a double-nested, JSON-encoded array of polygons. Each polygon is an array of points, where each point has x and y coordinates. Coordinates are relative to the PDF page, with the origin (0, 0) at the top-left corner. Currently, images extracted using `imageAction: generateNormalizedImages` will always produce a single polygon, but the double-nested structure is maintained for consistency with the Document Layout skill, which supports multiple polygons.

Define skillsets for image processing

This section supplements the [skill reference](#) articles by providing context for working with skill inputs, outputs, and patterns, as they relate to image processing.

1. [Create or update a skillset](#) to add skills.
2. Add templates for OCR and image analysis from the Azure portal, or copy the definitions from the [skill reference](#) documentation. Insert them into the skills array of your skillset definition.

3. If necessary, [include a Microsoft Foundry resource key](#) in the skillset. Azure AI Search makes calls to a billable Foundry resource for OCR and image analysis for transactions that exceed the free limit (20 per indexer per day). The Foundry resource must be in the same region as your search service.
4. If original images are embedded in PDF or application files like PPTX or DOCX, you need to add a Text Merge skill if you want image output and text output together. Working with embedded images is discussed further on in this article.

Once the basic framework of your skillset is created and Foundry Tools is configured, you can focus on each individual image skill, defining inputs and source context, and mapping outputs to fields in either an index or knowledge store.

 **Note**

For an example skillset that combines image processing with downstream natural language processing, see [REST Tutorial: Use REST and AI to generate searchable content from Azure blobs](#). It shows how to feed skill imaging output into entity recognition and key phrase extraction.

Inputs for image processing

As noted, images are extracted during document cracking and then normalized as a preliminary step. The normalized images are the inputs to any image processing skill, and are always represented in an enriched document tree in either one of two ways:

- `/document/normalized_images/*` is for documents that are processed whole.
- `/document/normalized_images/*/pages` is for documents that are processed in chunks (pages).

Whether you're using OCR and image analysis in the same, inputs have virtually the same construction:

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",  
  "context": "/document/normalized_images/*",  
  "detectOrientation": true,  
  "inputs": [  
    {  
      "name": "image",  
      "source": "/document/normalized_images/*"
```

```

        },
    ],
    "outputs": [ ]
},
{
    "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
    "context": "/document/normalized_images/*",
    "visualFeatures": [ "tags", "description" ],
    "inputs": [
        {
            "name": "image",
            "source": "/document/normalized_images/*"
        }
    ],
    "outputs": [ ]
}

```

Map outputs to search fields

In a skillset, Image Analysis and OCR skill output is always text. Output text is represented as nodes in an internal enriched document tree, and each node must be mapped to fields in a search index, or to projections in a knowledge store, to make the content available in your app.

1. In the skillset, review the `outputs` section of each skill to determine which nodes exist in the enriched document:

JSON

```

{
    "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
    "context": "/document/normalized_images/*",
    "detectOrientation": true,
    "inputs": [ ],
    "outputs": [
        {
            "name": "text",
            "targetName": "text"
        },
        {
            "name": "layoutText",
            "targetName": "layoutText"
        }
    ]
}

```

2. [Create or update a search index](#) to add fields to accept the skill outputs.

In the following fields collection example, *content* is blob content.

Metadata_storage_name contains the name of the file (set `retrievable` to *true*).

Metadata_storage_path is the unique path of the blob and is the default document key.
Merged_content is output from Text Merge (useful when images are embedded).

Text and *layoutText* are OCR skill outputs and must be a string collection in order to the capture all of the OCR-generated output for the entire document.

JSON

```
"fields": [
  {
    "name": "content",
    "type": "Edm.String",
    "filterable": false,
    "retrievable": true,
    "searchable": true,
    "sortable": false
  },
  {
    "name": "metadata_storage_name",
    "type": "Edm.String",
    "filterable": true,
    "retrievable": true,
    "searchable": true,
    "sortable": false
  },
  {
    "name": "metadata_storage_path",
    "type": "Edm.String",
    "filterable": false,
    "key": true,
    "retrievable": true,
    "searchable": false,
    "sortable": false
  },
  {
    "name": "merged_content",
    "type": "Edm.String",
    "filterable": false,
    "retrievable": true,
    "searchable": true,
    "sortable": false
  },
  {
    "name": "text",
    "type": "Collection(Edm.String)",
    "filterable": false,
    "retrievable": true,
    "searchable": true
  },
  {
    "name": "layoutText",
    "type": "Collection(Edm.String)",
    "filterable": false,
```

```
        "retrievable": true,  
        "searchable": true  
    },  
],
```

3. Update the indexer to map skillset output (nodes in an enrichment tree) to index fields.

Enriched documents are internal. To externalize the nodes in an enriched document tree, set up an output field mapping that specifies which index field receives node content.

Enriched data is accessed by your app through an index field. The following example shows a *text* node (OCR output) in an enriched document that's mapped to a *text* field in a search index.

JSON

```
"outputFieldMappings": [  
    {  
        "sourceFieldName": "/document/normalized_images/*/text",  
        "targetFieldName": "text"  
    },  
    {  
        "sourceFieldName": "/document/normalized_images/*/layoutText",  
        "targetFieldName": "layoutText"  
    }  
]
```

4. Run the indexer to invoke source document retrieval, image processing, and indexing.

Verify results

Run a query against the index to check the results of image processing. Use [Search Explorer](#) as a search client, or any tool that sends HTTP requests. The following query selects fields that contain the output of image processing.

HTTP

```
POST /indexes/[index name]/docs/search?api-version=[api-version]  
{  
    "search": "*",  
    "select": "metadata_storage_name, text, layoutText, imageCaption, imageTags"  
}
```

OCR recognizes text in image files. This means that OCR fields (*text* and *layoutText*) are empty if source documents are pure text or pure imagery. Similarly, image analysis fields (*imageCaption* and *imageTags*) are empty if source document inputs are strictly text. Indexer execution emits warnings if imaging inputs are empty. Such warnings are to be expected when nodes are unpopulated in the enriched document. Recall that blob indexing lets you include or exclude

file types if you want to work with content types in isolation. You can use these settings to reduce noise during indexer runs.

An alternate query for checking results might include the *content* and *merged_content* fields. Notice that those fields include content for any blob file, even those where there was no image processing performed.

About skill outputs

Skill outputs include `text` (OCR), `layoutText` (OCR), `merged_content`, `captions` (image analysis), `tags` (image analysis):

- `text` stores OCR-generated output. This node should be mapped to field of type `Collection(Edm.String)`. There's one `text` field per search document consisting of comma-delimited strings for documents that contain multiple images. The following illustration shows OCR output for three documents. First is a document containing a file with no images. Second is a document (image file) containing one word, *Microsoft*. Third is a document containing multiple images, some without any text ("").

JSON

```
"value": [
  {
    "@search.score": 1,
    "metadata_storage_name": "facts-about-microsoft.html",
    "text": []
  },
  {
    "@search.score": 1,
    "metadata_storage_name": "guthrie.jpg",
    "text": [ "Microsoft" ]
  },
  {
    "@search.score": 1,
    "metadata_storage_name": "Foundry Tools and Content Intelligence.pptx",
    "text": [
      "",
      "Microsoft",
      "",
      "",
      "",
      "",
      "Azure AI Search and Augmentation Combining Foundry Tools and Azure
Search"
    ]
  }
]
```

- `layoutText` stores OCR-generated information about text location on the page, described in terms of bounding boxes and coordinates of the normalized image. This node should be mapped to field of type `Collection(Edm.String)`. There's one `layoutText` field per search document consisting of comma-delimited strings.
- `merged_content` stores the output of a Text Merge skill, and it should be one large field of type `Edm.String` that contains raw text from the source document, with embedded `text` in place of an image. If files are text-only, then OCR and image analysis have nothing to do, and `merged_content` is the same as `content` (a blob property that contains the content of the blob).
- `imageCaption` captures a description of an image as individuals tags and a longer text description.
- `imageTags` stores tags about an image as a collection of keywords, one collection for all images in the source document.

The following screenshot is an illustration of a PDF that includes text and embedded images. Document cracking detected three embedded images: flock of seagulls, map, eagle. Other text in the example (including titles, headings, and body text) was extracted as text and excluded from image processing.

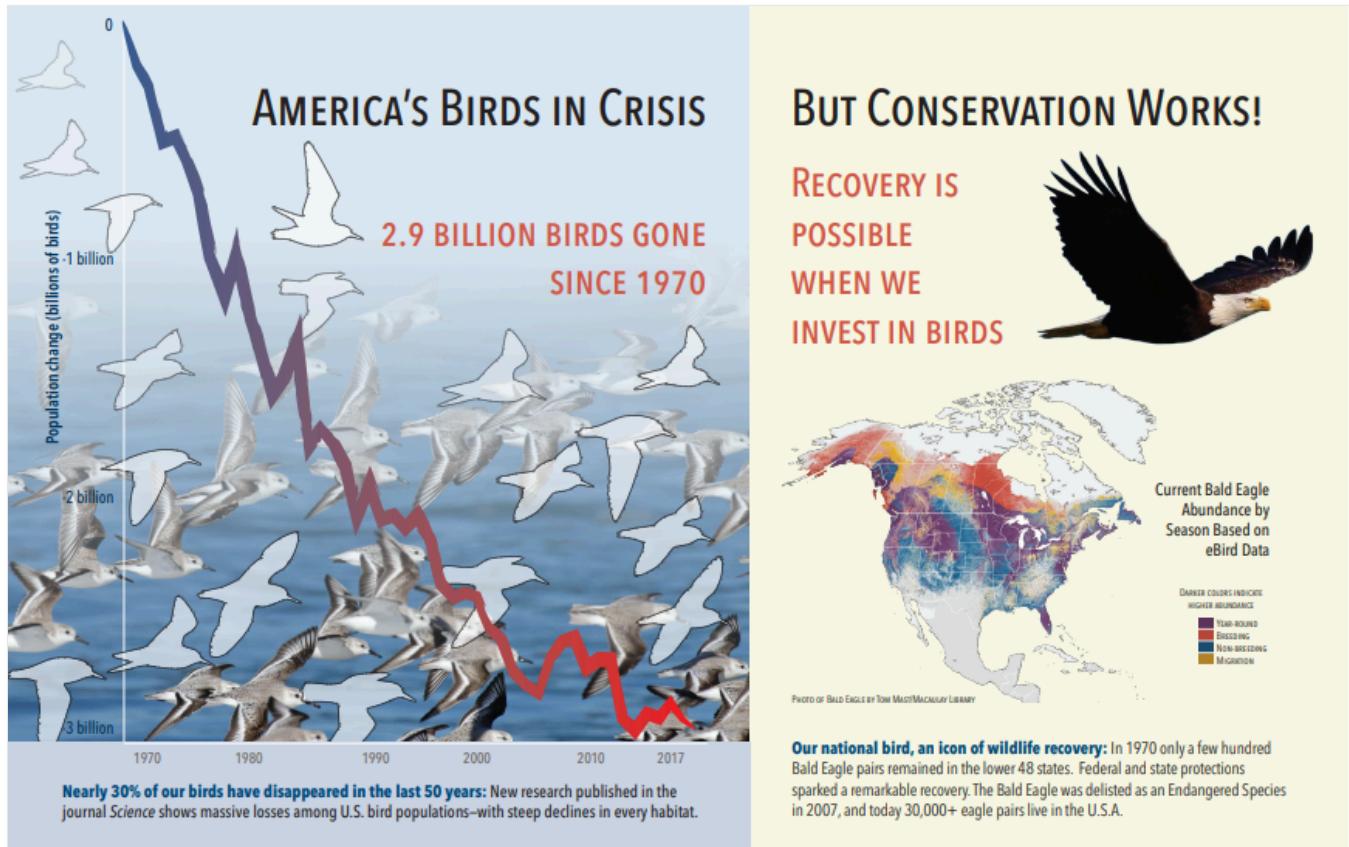


Image analysis output is illustrated in the following JSON (search result). The skill definition allows you to specify which [visual features](#) are of interest. For this example, tags and

descriptions were produced, but there are more outputs to choose from.

- `imageCaption` output is an array of descriptions, one per image, denoted by `tags` consisting of single words and longer phrases that describe the image. Notice the tags consisting of *a flock of seagulls are swimming in the water, or a close up of a bird*.
- `imageTags` output is an array of single tags, listed in the order of creation. Notice that tags repeat. There's no aggregation or grouping.

JSON

```
"imageCaption": [
    "{\"tags\": [
        \"bird\", \"outdoor\", \"water\", \"flock\", \"many\", \"lot\", \"bunch\", \"group\", \"several\",
        \"gathered\", \"pond\", \"lake\", \"different\", \"family\", \"flying\", \"standing\",
        \"little\", \"air\", \"beach\", \"swimming\", \"large\", \"dog\", \"landing\", \"jumping\",
        \"playing\"], \"captions\": [{\"text\": \"a flock of seagulls are swimming in the
        water\", \"confidence\": 0.70419257326275686}]}",
    "{\"tags\": [\"map\"], \"captions\": [
        {\"text\": \"map\", \"confidence\": 0.99942880868911743}]}",
    "{\"tags\": [
        \"animal\", \"bird\", \"raptor\", \"eagle\", \"sitting\", \"table\"], \"captions\": [
        {\"text\": \"a close up of a bird\", \"confidence\": 0.89643581933539462}]}",
    ...
],
"imageTags": [
    "bird",
    "outdoor",
    "water",
    "flock",
    "animal",
    "bunch",
    "group",
    "several",
    "drink",
    "gathered",
    "pond",
    "different",
    "family",
    "same",
    "map",
    "text",
    "animal",
    "bird",
    "bird of prey",
    "eagle"
]
}
```

Scenario: Embedded images in PDFs

When the images you want to process are embedded in other files, such as PDF or DOCX, the enrichment pipeline extracts just the images and then passes them to OCR or image analysis for processing. Image extraction occurs during the document cracking phase, and once the images are separated, they remain separate unless you explicitly merge the processed output back into the source text.

Text Merge is used to put image processing output back into the document. Although Text Merge isn't a hard requirement, it's frequently invoked so that image output (OCR text, OCR layoutText, image tags, image captions) can be reintroduced into the document. Depending on the skill, the image output replaces an embedded binary image with an in-place text equivalent. Image Analysis output can be merged at image location. OCR output always appears at the end of each page.

The following workflow outlines the process of image extraction, analysis, merging, and how to extend the pipeline to push image-processed output into other text-based skills such as Entity Recognition or Text Translation.

1. After connecting to the data source, the indexer loads and cracks source documents, extracting images and text, and queuing each content type for processing. An enriched document consisting only of a root node (*document*) is created.
2. Images in the queue are **normalized** and passed into enriched documents as a [`document/normalized_images`](#) node.
3. Image enrichments execute, using [`/document/normalized_images`](#) as input.
4. Image outputs are passed into the enriched document tree, with each output as a separate node. Outputs vary by skill (text and layoutText for OCR; tags and captions for Image Analysis).
5. Optional but recommended if you want search documents to include both text and image-origin text together, **Text Merge** runs, combining the text representation of those images with the raw text extracted from the file. Text chunks are consolidated into a single large string, where the text is inserted first in the string and then the OCR text output or image tags and captions.

The output of Text Merge is now the definitive text to analyze for any downstream skills that perform text processing. For example, if your skillset includes both OCR and Entity Recognition, the input to Entity Recognition should be [`"document/merged_text"`](#) (the `targetName` of the Text Merge skill output).

6. After all skills have executed, the enriched document is complete. In the last step, indexers refer to **output field mappings** to send enriched content to individual fields in the search index.

The following example skillset creates a `merged_text` field containing the original text of your document with embedded OCRed text in place of embedded images. It also includes an Entity Recognition skill that uses `merged_text` as input.

Request body syntax

JSON

```
{
  "description": "Extract text from images and merge with content text to produce merged_text",
  "skills": [
    {
      "description": "Extract text (plain and structured) from image.",
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
      "context": "/document/normalized_images/*",
      "defaultLanguageCode": "en",
      "detectOrientation": true,
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [
        {
          "name": "text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
      "description": "Create merged_text, which includes all the textual representation of each image inserted at the right location in the content field.",
      "context": "/document",
      "insertPreTag": " ",
      "insertPostTag": " ",
      "inputs": [
        {
          "name": "text", "source": "/document/content"
        },
        {
          "name": "itemsToInsert", "source": "/document/normalized_images/*/text"
        },
        {
          "name": "offsets", "source": "/document/normalized_images/*/contentOffset"
        }
      ],
      "outputs": [
        {
          "name": "mergedText", "targetName" : "merged_text"
        }
      ]
    }
  ]
}
```



```

        double rotationFromOriginal)
{
    Point original = new Point();
    double angle = rotationFromOriginal % 360;

    if (angle == 0 )
    {
        original.X = normalized.X;
        original.Y = normalized.Y;
    } else if (angle == 90)
    {
        original.X = normalized.Y;
        original.Y = (width - normalized.X);
    } else if (angle == 180)
    {
        original.X = (width - normalized.X);
        original.Y = (height - normalized.Y);
    } else if (angle == 270)
    {
        original.X = height - normalized.Y;
        original.Y = normalized.X;
    }

    double scalingFactor = (angle % 180 == 0) ? originalHeight / height :
originalHeight / width;
    original.X = (int) (original.X * scalingFactor);
    original.Y = (int)(original.Y * scalingFactor);

    return original;
}

```

Scenario: Custom image skills

Images can also be passed into and returned from custom skills. A skillset base64-encodes the image being passed into the custom skill. To use the image within the custom skill, set `"/document/normalized_images/*/data"` as the input to the custom skill. Within your custom skill code, base64-decode the string before converting it to an image. To return an image to the skillset, base64-encode the image before returning it to the skillset.

The image is returned as an object with the following properties.

JSON

```
{
    "$type": "file",
    "data": "base64String"
}
```

The [Azure Search Python samples](#) repository has a complete sample implemented in Python of a custom skill that enriches images.

Passing images to custom skills

For scenarios where you require a custom skill to work on images, you can pass images to the custom skill, and have it return text or images. The following skillset is from a sample.

The following skillset takes the normalized image (obtained during document cracking), and outputs slices of the image.

Sample skillset

JSON

```
{  
    "description": "Extract text from images and merge with content text to produce merged_text",  
    "skills":  
    [  
        {  
            "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
            "name": "ImageSkill",  
            "description": "Segment Images",  
            "context": "/document/normalized_images/*",  
            "uri": "https://your.custom.skill.url",  
            "httpMethod": "POST",  
            "timeout": "PT30S",  
            "batchSize": 100,  
            "degreeOfParallelism": 1,  
            "inputs": [  
                {  
                    "name": "image",  
                    "source": "/document/normalized_images/*"  
                }  
            ],  
            "outputs": [  
                {  
                    "name": "slices",  
                    "targetName": "slices"  
                }  
            ],  
            "httpHeaders": {}  
        }  
    ]  
}
```

Custom skill example

The custom skill itself is external to the skillset. In this case, it's Python code that first loops through the batch of request records in the custom skill format, then converts the base64-encoded string to an image.

Python

```
# deserialize the request, for each item in the batch
for value in values:
    data = value['data']
    base64String = data["image"]["data"]
    base64Bytes = base64String.encode('utf-8')
    inputBytes = base64.b64decode(base64Bytes)
    # Use numpy to convert the string to an image
    jpg_as_np = np.frombuffer(inputBytes, dtype=np.uint8)
    # you now have an image to work with
```

Similarly to return an image, return a base64 encoded string within a JSON object with a `$type` property of *file*.

Python

```
def base64EncodeImage(image):
    is_success, im_buf_arr = cv2.imencode(".jpg", image)
    byte_im = im_buf_arr.tobytes()
    base64Bytes = base64.b64encode(byte_im)
    base64String = base64Bytes.decode('utf-8')
    return base64String

base64String = base64EncodeImage(jpg_as_np)
result = {
    "$type": "file",
    "data": base64String
}
```

Related content

- [Create indexer \(REST\)](#)
- [Image Analysis skill](#)
- [OCR skill](#)
- [Text Merge skill](#)
- [How to create a skillset](#)
- [Map enriched output to fields](#)

Configure an enrichment cache

08/27/2025

ⓘ Important

This feature is in public preview under [supplemental terms of use](#). [Preview REST APIs](#) support this feature.

This article explains how to add caching to an enrichment pipeline so that you can modify downstream enrichment steps without having to rebuild in full every time. By default, a skillset is stateless, and changing any part of its composition requires a full rerun of the indexer. With an *enrichment cache*, the indexer can determine which parts of the document tree must be refreshed based on changes detected in the skillset or indexer definitions. Existing processed output is preserved and reused wherever possible.

Cached content is placed in Azure Storage using account information that you provide. The container, named `ms-az-search-indexercache-<alpha-numerical-string>`, is created when you run the indexer. It should be considered an internal component managed by your search service and must not be modified.

Prerequisites

- [Azure Storage](#) for storing cached enrichments. The storage account must be [general purpose v2](#).
- [For blob indexing only](#), if you need synchronized document removal from both the cache and index when blobs are deleted from your data source, enable a [deletion policy](#) in the indexer. Without this policy, document deletion from the cache isn't supported.

You should be familiar with setting up indexers and skillsets. Start with [indexer overview](#) and then continue on to [skillsets](#) to learn about enrichment pipelines.

Limitations

ⓘ Caution

If you're using the [SharePoint Online indexer \(Preview\)](#), you should avoid incremental enrichment. Under certain circumstances, the cache becomes invalid, requiring an [indexer reset and full rebuild](#), should you choose to reload it.

Permissions

Azure AI Search needs write-access to Azure Storage. If you're using a managed identity for your search service, make sure it's assigned to the **Storage Blob Data Contributor** and **Storage Table Data Contributor** roles. For more information, see [Connect to Azure Storage using a managed identity \(Azure AI Search\)](#).

Enable on new indexers

You can use the Azure portal, preview APIs, or preview Azure SDK packages to enable an enrichment cache on an indexer.

Azure portal

The screenshot shows the 'Add indexer' configuration screen in the Azure portal. It includes fields for 'Enable incremental enrichment' (checked) and 'Indexer cache location' (containing a connection string). A callout highlights the 'Indexer cache location' field, which contains a purple URL and a 'Choose an existing connection' link.

1. On the left, select **Indexers**, and then select **Add indexer**.
2. Provide an indexer name and an existing index, data source, and skillset.
3. Enable incremental caching and set the Azure Storage account.

Enable on existing indexers

For existing indexers that already have a skillset, use the following steps to add caching. As a one-time operation, reset and rerun the indexer in full to load the cache.

Step 1: Get the indexer definition

Start with a valid, work indexer that has these components: data source, skillset, index. Using an API client, send a [GET Indexer](#) request to retrieve the indexer. When you use the preview API version to the GET the indexer, a `cache` property set to null is added to the definition automatically.

HTTP

```
GET https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?
api-version=2025-08-01-preview
Content-Type: application/json
api-key: [YOUR-ADMIN-KEY]
```

Step 2: Set the cache property

In the index definition, modify `cache` to include the following required and optional properties:

- (Required) `storageConnectionString` must be set to an Azure Storage connection string.
- (Optional) `enableReprocessing` boolean property (`true` by default), indicates that incremental enrichment is enabled. Set to `false` if you want to suspend incremental processing while other resource-intensive operations, such as indexing new documents, are underway and then switch back to `true` later.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-08-01-
preview
{
    "name": "<YOUR-INDEXER-NAME>",
    "targetIndexName": "<YOUR-INDEX-NAME>",
    "dataSourceName": "<YOUR-DATASOURCE-NAME>",
    "skillsetName": "<YOUR-SKILLSET-NAME>",
    `cache` : {
        "storageConnectionString" : "<YOUR-STORAGE-ACCOUNT-CONNECTION-
STRING>",
        "enableReprocessing": true
    },
    "fieldMappings" : [],
    "outputFieldMappings": [],
    "parameters": []
}
```

Step 3: Reset the indexer

[Reset Indexer](#) is required when setting up incremental enrichment for existing indexers to ensure all documents are in a consistent state. You can use the Azure portal or an API client for this task.

https

```
POST https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-
NAME]/reset?api-version=2025-08-01-preview
```

```
Content-Type: application/json  
api-key: [YOUR-ADMIN-KEY]
```

Step 4: Save the indexer

Update Indexer with a PUT request, where the body of the request includes `cache`.

HTTP

```
PUT https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?  
api-version=2025-08-01-preview  
Content-Type: application/json  
api-key: [YOUR-ADMIN-KEY]  
{  
    "name" : "<YOUR-INDEXER-NAME>",  
    ...  
    `cache` : {  
        "storageConnectionString": "<YOUR-STORAGE-ACCOUNT-CONNECTION-STRING>",  
        "enableReprocessing": true  
    }  
}
```

If you now issue another GET request on the indexer, the response from the service includes an `ID` property in the `cache` object. The string is appended to the name of the container containing all the cached results and intermediate state of each document processed by this indexer. The ID is used to uniquely name the cache in Blob storage.

HTTP

```
`cache` : {  
    "ID": "<ALPHA-NUMERIC STRING>",  
    "enableReprocessing": true,  
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<YOUR-STORAGE-ACCOUNT>;AccountKey=<YOUR-STORAGE-KEY>;EndpointSuffix=core.windows.net"  
}
```

Step 5: Run the indexer

To run indexer, you can use the Azure portal or the API. In the Azure portal, from the indexers list, select the indexer and select **Run**. One advantage to using the Azure portal is that you can monitor indexer status, note the duration of the job, and how many documents are processed. Portal pages are refreshed every few minutes.

Alternatively, you can use REST to [run the indexer](#):

HTTP

```
POST https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-  
NAME]/run?api-version=2025-08-01-preview  
Content-Type: application/json  
api-key: [YOUR-ADMIN-KEY]
```

! Note

A reset and rerun of the indexer results in a full rebuild so that content can be cached. All cognitive enrichments will be rerun on all documents. Reusing enriched content from the cache begins after the cache is loaded.

Check for cached output

Find the cache in Azure Storage, under Blob container. The container name is `ms-az-search-indexercache-<some-alphanumeric-string>`.

A cache is created and used by an indexer. Its content isn't human readable.

To verify whether the cache is operational, modify a skillset and run the indexer, then compare before-and-after metrics for execution time and document counts.

Skillsets that include image analysis and Optical Character Recognition (OCR) of scanned documents make good test cases. If you modify a downstream text skill or any skill that isn't image-related, the indexer can retrieve all of the previously processed image and OCR content from cache, updating and processing only the text-related changes indicated by your edits. You can expect to see fewer documents in the indexer execution document count, shorter execution times, and fewer charges on your bill.

The [file set](#) used in [cog-search-demo tutorials](#) is a useful test case because it contains 14 files of various formats JPG, PNG, HTML, DOCX, PPTX, and other types. Change `en` to `es` or another language in the text translation skill for proof-of-concept testing of incremental enrichment.

Common errors

The following error occurs if you forget to specify a preview API version on the request:

```
"The request is invalid. Details: indexer : A resource without a type name was found, but  
no expected type was specified. To allow entries without type information, the expected  
type must also be specified when the model is specified."
```

A 400 Bad Request error will also occur if you're missing an indexer requirement. The error message specifies any missing dependencies.

Next step

Incremental enrichment is applicable on indexers that contain skillsets, providing reusable content for both indexes and knowledge stores. The following link provides more information about cache management.

- ✓ [Manage an enrichment cache](#)

Manage an enrichment cache

08/27/2025

ⓘ Important

This feature is in public preview under [supplemental terms of use](#). The [preview REST API](#) supports this feature.

An *enrichment cache* is an optional feature that stores reusable enriched content created during [skillset execution](#) so that only new and changed skills and documents incur standard processing charges during future indexer and skillset processing.

The cache contains the output from [document cracking](#), plus the outputs of each skill for every document. Although caching is billable (it uses Azure Storage), the overall cost of enrichment is reduced because the costs of storage are less than image extraction and AI processing.

If you have configured an enrichment cache, this article explains how to manage skill and data source updates so that you get maximum utility from cached enrichments.

Prerequisites

- An [indexer](#) and [skillset](#)
- An [enrichment cache](#)

Limitations

✗ Caution

If you're using the [SharePoint Online indexer \(Preview\)](#), you should avoid incremental enrichment. Under certain circumstances, the cache becomes invalid, requiring an [indexer reset and full rebuild](#), should you choose to reload it.

Cache configuration

Physically, the cache is stored in a blob container or table in your Azure Storage account, one per indexer. Each indexer is assigned a unique and immutable cache identifier that corresponds to the container it's using.

The cache is created when you specify the "cache" property and run the indexer. Only enriched content can be cached. If your indexer doesn't have an attached skillset, then caching doesn't apply.

The following example illustrates an indexer with caching enabled. See [Configure enrichment caching](#) for full instructions.

To set the cache property, use latest preview REST API for [Create or Update Indexer](#) or a preview Azure SDK package that provides the feature. You can also enable enrichment caching in the Import data wizard in the Azure portal.

JSON

```
POST https://[YOUR-SEARCH-SERVICE-NAME].search.windows.net/indexers?api-version=2025-08-01-preview
{
    "name": "myIndexerName",
    "targetIndexName": "myIndex",
    "dataSourceName": "myDatasource",
    "skillsetName": "mySkillset",
    "cache" : {
        "storageConnectionString" : "<Your storage account connection string>",
        "enableReprocessing": true
    },
    "fieldMappings" : [],
    "outputFieldMappings": [],
    "parameters": []
}
```

Cache management

The lifecycle of the cache is managed by the indexer. If an indexer is deleted, its cache is also deleted. If the `cache` property on the indexer is set to null or the connection string is changed, the existing cache is deleted on the next indexer run.

While incremental enrichment is designed to detect and respond to changes with no intervention on your part, there are parameters you can use to invoke specific behaviors:

- [Prioritize new documents](#)
- [Bypass skillset checks](#)
- [Bypass data source checks](#)
- [Force skillset evaluation](#)

Prioritize new documents

The cache property includes an `enableReprocessing` parameter. It's used to control processing over incoming documents already represented in the cache. When true (default), documents already in the cache are reprocessed when you rerun the indexer, assuming your skill update affects that doc.

When false, existing documents aren't reprocessed, effectively prioritizing new, incoming content over existing content. You should only set `enableReprocessing` to false on a temporary basis. Having `enableReprocessing` set to true most of the time ensures that all documents, both new and existing, are valid per the current skillset definition.

Bypass skillset evaluation

Modifying a skill and reprocessing of that skill typically go hand in hand. However, some changes to a skill shouldn't result in reprocessing (for example, deploying a custom skill to a new location or with a new access key). Most likely, these are peripheral modifications that have no genuine impact on the substance of the skill output itself.

If you know that a change to the skill is indeed superficial, you should override skill evaluation by setting the `disableCacheReprocessingChangeDetection` parameter to true:

1. Call [Update Skillset](#) and modify the skillset definition.
2. Append the "disableCacheReprocessingChangeDetection=true" parameter on the request.
3. Submit the change.

Setting this parameter ensures that only updates to the skillset definition are committed and the change isn't evaluated for effects on the existing cache. Use a preview API version, 2020-06-30-Preview or later. We recommend the latest preview API.

HTTP

```
PUT https://[servicename].search.windows.net/skillsets/[skillset name]?api-version=2025-08-01-preview&disableCacheReprocessingChangeDetection
```

Bypass data source validation checks

Most changes to a data source definition will invalidate the cache. However, for scenarios where you know that a change shouldn't invalidate the cache - such as changing a connection string or rotating the key on the storage account - append the `ignoreResetRequirement` parameter on the [data source update](#). Setting this parameter to true allows the commit to go

through, without triggering a reset condition that would result in all objects being rebuilt and populated from scratch.

HTTP

```
PUT https://[search service].search.windows.net/datasources/[data source name]?
api-version=2025-08-01-preview&ignoreResetRequirement
```

Force skillset evaluation

The purpose of the cache is to avoid unnecessary processing, but suppose you make a change to a skill that the indexer doesn't detect (for example, changing something in external code, such as a custom skill).

In this case, you can use the [Reset Skills](#) to force reprocessing of a particular skill, including any downstream skills that have a dependency on that skill's output. This API accepts a POST request with a list of skills that should be invalidated and marked for reprocessing. After Reset Skills, follow with a [Run Indexer](#) request to invoke the pipeline processing.

Re-cache specific documents

[Resetting an indexer](#) will result in all documents in the search corpus being reprocessed.

In scenarios where only a few documents need to be reprocessed, use [Reset Documents \(preview\)](#) to force reprocessing of specific documents. When a document is reset, the indexer invalidates the cache for that document, which is then reprocessed by reading it from the data source. For more information, see [Run or reset indexers, skills, and documents](#).

To reset specific documents, the request provides a list of document keys as read from the search index. If the key is mapped to a field in the external data source, the value that you provide should be the one used in the search index.

Depending on how you call the API, the request will either append, overwrite, or queue up the key list:

- Calling the API multiple times with different keys appends the new keys to the list of document keys reset.
- Calling the API with the "overwrite" query string parameter set to true will overwrite the current list of document keys to be reset with the request's payload.

- Calling the API only results in the document keys being added to the queue of work the indexer performs. When the indexer is next invoked, either as scheduled or on demand, it will prioritize processing the reset document keys before any other changes from the data source.

The following example illustrates a reset document request:

HTTP

```
POST https://[search service name].search.windows.net/indexers/[indexer  
name]/resetdocs?api-version=2025-08-01-preview  
{  
    "documentKeys" : [  
        "key1",  
        "key2",  
        "key3"  
    ]  
}
```

Changes that invalidate the cache

Once you enable a cache, the indexer evaluates changes in your pipeline composition to determine which content can be reused and which needs reprocessing. This section enumerates changes that invalidate the cache outright, followed by changes that trigger incremental processing.

An invalidating change is one where the entire cache is no longer valid. An example of an invalidating change is one where your data source is updated. Here's the complete list of changes to any part of the indexer pipeline that would invalidate your cache:

- Changing the data source type
- Changing data source container
- Changing data source credentials
- Changing data source change detection policy
- Changing data source delete detection policy
- Changing indexer field mappings
- Changing indexer parameters:
 - Parsing Mode
 - Excluded File Name Extensions
 - Indexed File Name Extensions
 - Index storage metadata only for oversized documents
 - Delimited text headers
 - Delimited text delimiter

- Document Root
- Image Action (Changes to how images are extracted)

Changes that trigger incremental processing

Incremental processing evaluates your skillset definition and determines which skills to rerun, selectively updating the affected portions of the document tree. Here's the complete list of changes resulting in incremental enrichment:

- Changing the skill type (the OData type of the skill is updated)
- Skill-specific parameters are updated, for example a URL, defaults, or other parameters
- Skill output changes, the skill returns additional or different outputs
- Skill input changes resulting in different ancestry, skill chaining has changed
- Any upstream skill invalidation, if a skill that provides an input to this skill is updated
- Updates to the knowledge store projection location, results in re-projecting documents
- Changes to the knowledge store projections, results in re-projecting documents
- Output field mappings changed on an indexer results in re-projecting documents to the index

APIs used for caching

Preview APIs provide extra properties on indexers. We recommend the latest preview API.

Skillsets and data sources can use the generally available version. In addition to the reference documentation, see [Configure caching for incremental enrichment](#) for details about order of operations.

- [Create or Update Indexer \(api-version=2025-08-01-preview\)](#)
- [Reset Skills \(api-version=2025-08-01-preview\)](#)
- [Create or Update Skillset \(api-version=2025-08-01-preview\)](#) (New URI parameter on the request)
- [Create or Update Data Source \(api-version=2025-08-01-preview\)](#), when called with a preview API version, provides a new parameter named "ignoreResetRequirement", which should be set to true when your update action shouldn't invalidate the cache. Use "ignoreResetRequirement" sparingly as it could lead to unintended inconsistency in your data that won't be detected easily.

Generate captions for images in another language

07/28/2025

In this article, learn how to generate captions using AI enrichment and a skillset. Images often contain useful information that's relevant in search scenarios. You can [vectorize images](#) to represent visual content in your search index. Or, you can use [AI enrichment and skillsets](#) to create and extract searchable *text* from images.

The GenAI Prompt skill (preview) generates a description of each image in your data source and the indexer pushes that description into a search index. To view the descriptions, you can run a query that includes them in the response.

Prerequisites

To work with image content in a skillset, you need:

- A [supported data source](#). We recommend Azure Storage.
- Files or blobs containing images.
- Read access to the supported data source. This article uses key-based authentication, but indexers can also connect using the search service identity and Microsoft Entra ID authentication. For role-based access control, assign roles on the data source to allow read access by the service identity. If you're testing on a local development machine, make sure you also have read access on the supported data source.
- A [search indexer](#), configured for image actions.
- A skillset with the new custom genAI prompt skill.
- A search index with fields to receive the verbalized text output, plus output field mappings in the indexer that establish association.

Optionally, you can define projections to accept image-analyzed output into a [knowledge store](#) for data mining scenarios.

Configure indexers for image processing

After the source files are set up, enable image normalization by setting the `imageAction` parameter in the indexer configuration. Image normalization helps make images more uniform for downstream processing. Image normalization includes the following operations:

- Large images are resized to a maximum height and width to make them uniform.

- For images that have metadata that specifies orientation, image rotation is adjusted for vertical loading.

Note that enabling `imageAction` (setting this parameter to other than `none`) will incur an additional charge for image extraction according to [Azure AI Search pricing](#).

1. [Create or update an indexer](#) to set the configuration properties:

```
JSON

{
  "parameters": {
    "configuration": {
      "dataToExtract": "contentAndMetadata",
      "parsingMode": "default",
      "imageAction": "generateNormalizedImages"
    }
  }
}
```

2. Set `dataToExtract` to `contentAndMetadata` (required).

3. Verify that the `parsingMode` is set to `default` (required).

This parameter determines the granularity of search documents created in the index. The default mode sets up a one-to-one correspondence so that one blob results in one search document. If documents are large, or if skills require smaller chunks of text, you can add the Text Split skill that subdivides a document into paging for processing purposes. But for search scenarios, one blob per document is required if enrichment includes image processing.

4. Set `imageAction` to enable the `normalized_images` node in an enrichment tree (required):

- `generateNormalizedImages` to generate an array of normalized images as part of document cracking.
- `generateNormalizedImagePerPage` (applies to PDF only) to generate an array of normalized images where each page in the PDF is rendered to one output image. For non-PDF files, the behavior of this parameter is similar as if you had set `generateNormalizedImages`. However, setting `generateNormalizedImagePerPage` can make indexing operation less performant by design (especially for large documents) since several images would have to be generated.

5. Optionally, adjust the width or height of the generated normalized images:

- `normalizedImageMaxWidth` in pixels. Default is 2,000. Maximum value is 10,000.
- `normalizedImageMaxHeight` in pixels. Default is 2,000. Maximum value is 10,000.

About normalized images

When `imageAction` is set to a value other than `none`, the new `normalized_images` field contains an array of images. Each image is a complex type that has the following members:

 [Expand table](#)

Image member	Description
<code>data</code>	BASE64 encoded string of the normalized image in JPEG format.
<code>width</code>	Width of the normalized image in pixels.
<code>height</code>	Height of the normalized image in pixels.
<code>originalWidth</code>	The original width of the image before normalization.
<code>originalHeight</code>	The original height of the image before normalization.
<code>rotationFromOriginal</code>	Counter-clockwise rotation in degrees that occurred to create the normalized image. A value between 0 degrees and 360 degrees. This step reads the metadata from the image that is generated by a camera or scanner. Usually a multiple of 90 degrees.
<code>contentOffset</code>	The character offset within the content field where the image was extracted from. This field is only applicable for files with embedded images. The <code>contentOffset</code> for images extracted from PDF documents is always at the end of the text on the page it was extracted from in the document. This means images appear after all text on that page, regardless of the original location of the image in the page.
<code>pageNumber</code>	If the image was extracted or rendered from a PDF, this field contains the page number in the PDF it was extracted or rendered from, starting from 1. If the image isn't from a PDF, this field is 0.

Sample value of `normalized_images`:

JSON

```
[  
  {  
    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",  
    "width": 500,  
    "height": 300,  
    "originalWidth": 5000,  
    "originalHeight": 3000,  
  }]
```

```
        "rotationFromOriginal": 90,  
        "contentOffset": 500,  
        "pageNumber": 2  
    }  
]
```

Define skillsets for image processing

This section supplements the [skill reference](#) articles by providing context for working with skill inputs, outputs, and patterns, as they relate to image processing.

- Create or update a skillset to add skills.

Once the basic framework of your skillset is created and Azure AI services is configured, you can focus on each individual image skill, defining inputs and source context, and mapping outputs to fields in either an index or knowledge store.

 **Note**

For an example skillset that combines image processing with downstream natural language processing, see [REST Tutorial: Use REST and AI to generate searchable content from Azure blobs](#). It shows how to feed skill imaging output into entity recognition and key phrase extraction.

Example inputs for image processing

As noted, images are extracted during document cracking and then normalized as a preliminary step. The normalized images are the inputs to any image processing skill, and are always represented in an enriched document tree in either one of two ways:

- `/document/normalized_images/*` is for documents that are processed whole.

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",  
    "context": "/document/normalized_images/*",  
    "uri": "https://contoso.openai.azure.com/openai/deployments/contoso-gpt-  
4o/chat/completions?api-version=2025-01-01-preview",  
    "timeout": "PT1M",  
    "apiKey": "<YOUR-API-KEY here>"  
    "inputs": [  
        {  
            "name": "image",  
            "type": "Image"  
        }  
    ]  
}
```

```

        "source": "/document/normalized_images/*/data"
    },
    {
        "name": "systemMessage",
        "source": "'You are a useful artificial intelligence assistant that
helps people.'"
    },
    {
        "name": "userMessage",
        "source": "'Describe what you see in this image in 20 words or less in
Spanish.'"
    }
],
"outputs": [
{
    "name": "response",
    "targetName": "captionedImage"
}
]
},

```

Example using json schema responses with text inputs

This example illustrates how you can use structured outputs for language models. Note that this capability is mainly supported mostly by OpenAI language models, although that may change in the future.

JSON

```

{
    "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",
    "context": "/document/content",
    "uri": "https://contoso.openai.azure.com/openai/deployments/contoso-gpt-
4o/chat/completions?api-version=2025-01-01-preview",
    "timeout": "PT1M",
    "apiKey": "<YOUR-API-KEY here>"
    "inputs": [
        {
            "name": "systemMessage",
            "source": "'You are a useful artificial intelligence assistant that
helps people.'"
        },
        {
            "name": "userMessage",
            "source": "'How many languages are there in the world and what are
they?'"
        }
    ],
    "response_format": {
        "type": "json_schema",
        "json_schema": {

```

```

    "name": "structured_output",
    "strict": true,
    "schema": {
        "type": "object",
        "properties": {
            "total": { "type": "number" },
            "languages": {
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": ["total", "languages"],
        "additionalProperties": false
    }
},
"outputs": [
{
    "name": "response",
    "targetName": "responseJsonForLanguages"
}
]
},

```

Map outputs to search fields

Output text is represented as nodes in an internal enriched document tree, and each node must be mapped to fields in a search index, or to projections in a knowledge store, to make the content available in your app.

1. [Create or update a search index](#) to add fields to accept the skill outputs.

In the following fields collection example, *content* is blob content.

Metadata_storage_name contains the name of the file (set `retrievable` to *true*).

Metadata_storage_path is the unique path of the blob and is the default document key.

Merged_content is output from Text Merge (useful when images are embedded).

captionedImage is the skill outputs and must be a string-type in order to capture all of the language model output in the search index.

JSON

```

"fields": [
{
    "name": "content",
    "type": "Edm.String",
    "filterable": false,
    "retrievable": true,

```

```

        "searchable": true,
        "sortable": false
    },
{
    "name": "metadata_storage_name",
    "type": "Edm.String",
    "filterable": true,
    "retrievable": true,
    "searchable": true,
    "sortable": false
},
{
    "name": "metadata_storage_path",
    "type": "Edm.String",
    "filterable": false,
    "key": true,
    "retrievable": true,
    "searchable": false,
    "sortable": false
},
{
    "name": "captioned_image",
    "type": "Edm.String",
    "filterable": false,
    "retrievable": true,
    "searchable": true,
    "sortable": false
}
]

```

2. Update the indexer to map skillset output (nodes in an enrichment tree) to index fields.

Enriched documents are internal. To externalize the nodes in an enriched document tree, set up an output field mapping that specifies which index field receives node content. Enriched data is accessed by your app through an index field. The following example shows a *text* node (OCR output) in an enriched document that's mapped to a *text* field in a search index.

JSON

```

"outputFieldMappings": [
{
    "sourceFieldName": "/document/normalized_images/*/captionedImage",
    "targetFieldName": "captioned_image"
}
]

```

3. Run the indexer to invoke source document retrieval, image processing via language model captions, and indexing.

Verify results

Run a query against the index to check the results of image processing. Use [Search Explorer](#) as a search client, or any tool that sends HTTP requests. The following query selects fields that contain the output of image processing.

HTTP

```
POST /indexes/[index name]/docs/search?api-version=[api-version]
{
  "search": "A cat in a picture",
  "select": "metadata_storage_name, captioned_image"
}
```

Related content

- [Create indexer \(REST\)](#)
- [GenAI Prompt skill](#)
- [How to create a skillset](#)
- [Map enriched output to fields](#)

Best practices - GenAI Prompt skill

07/28/2025

Incorporating the GenAI Prompt custom skill as part of an indexer's ingestion flow allows developers to harness the content generation capabilities of language models to enrich the content from the data source. This document outlines some recommendations and best practices that can be incorporated when utilizing this capability to ensure good system performance and behavior.

Functionality of GenAI Prompt skill

The [GenAI Prompt custom skill](#) is a new addition to Azure AI Search's catalog of skills, allowing search customers to pass their document contents and customer-created prompts to a language model they own, hosted within Azure AI Foundry. The resulting enriched content, along with the source document is ingested into the search index. We envision developers utilizing this for various scenarios to power Retrieval Augmented Generation (RAG) applications. Some key scenarios are AI-generated document summaries, image captioning, and entity/fact finding based on customizable criteria through user specified prompts.

The content generation capabilities of language models are continuing to evolve rapidly and their integration into content ingestion pipeline offers exciting possibilities for search retrieval relevance. However, the challenge for developers would be to ensure that the prompts and data they use for their scenarios are safe and protect users against any inadvertent results from the language model.

Personas interacting with a RAG application

In order to list out the various challenges in incorporating AI content generation capabilities into an Azure AI Search indexer pipeline, it's important to understand the various personas that interact with the RAG application as each of them might carry a different set of challenges.

 Expand table

Persona	Description
End user	The person asking questions of the RAG application, expecting a well-cited answer to their question based on results from the source document. In addition to accuracy of the answer, the end-user expects that any citations provided by the application make it clear if it was from verbatim content from a source file or an AI-powered summary from the model.

Persona	Description
RAG application developer/search index admin	<p>The person responsible for configuring the search index schema, and setting up the indexer and skillset to ingest language model augmented data into the index. GenAI Prompt custom skill allows developers to configure free-form prompts to several models hosted in AI foundry, thereby offering significant flexibility to light up various scenarios. However, developers need to ensure that the combination of data and skills used in the pipeline doesn't produce harmful or unsafe content. Developers also need to evaluate the content generated by the language models for bias, inaccuracies, and incorrect information. Although this task can be challenging for documents at a large scale, it should be one of the first steps when building a RAG application, along with the index schema definition.</p>
Data authority	<p>The person expected to be the key subject matter expert (SME) for the content from the data source. The SME is expected to be the best judge of language model powered enrichments ingested into the index and the answer generated by the language model in the RAG application. The key role for the data authority to be able to get a representative sample and verify the quality of the enrichments and the answer, which can be challenging if dealing with data at large scale.</p>

The rest of this document lists out these various challenges along with tips and best practices that RAG application developers can follow to mitigate any risks.

Challenges

The following challenges are faced by the various personas that interact with a RAG systems that utilize language models to augment content ingested into a search index (using the GenAI Prompt custom skill) and to formulate answers for questions:

- Transparency: Users of RAG systems should understand that AI models might not always produce accurate or well-formulated answers. Azure AI Search has a robustly documented [Transparency Note](#) that developers should read through to understand the various ways in which AI is used to augment the capabilities of the core search engine. It's recommended that developers who build RAG applications share the transparency note to users of their applications, since they might be unaware of how AI interfaces with various aspects of the application being used. Additionally, when utilizing the GenAI Prompt custom skill developers should note that only part of the content ingested into the search index is generated by the language model and should highlight this to users of their applications.
- Content sampling/inspection of content quality: Developers and data SMEs should consider sampling some of the content ingested into the search index after being augmented by the GenAI Prompt custom skill in order to inspect the quality of the

enrichment performed by their language model. [Debug sessions](#) and [search explorer](#) on the Azure portal can be used for this purpose.

- Content safety filtering and evaluations: It's important for developers to ensure that the language models they use with the GenAI Prompt custom skill have appropriate filters to ensure safety of the content generated and after ingested into the search index. Developers and data SMEs should also make sure they evaluate the content generated by the language model on various metrics such as accuracy, task specific performance, bias, and risk. Azure AI Foundry offers a robust set of tools for developers to add [content safety filters](#) and [clear guidance for evaluation approaches](#).
- Agility in rolling back changes or modifying skill configuration: It's possible for the language model used with the GenAI Prompt custom skill to have issues over time (such as producing low-quality content). Developers should be prepared to roll back these changes either by altering their indexer and skillset configuration or by excluding index fields with AI generated content from search queries.

Best practices to mitigate risks

When utilizing the GenAI Prompt custom skill to power RAG applications, there's a risk of over-reliance on AI as outlined in the challenges from the previous section. In this part of the document, we present some patterns and strategies to use to mitigate the risks and overcome the challenges.

Content sampling and inspection before ingestion into the search index

[Debug sessions](#) is a tool built into the Azure portal. You can use it to inspect the state of enrichment for a single document. To start a debug session, create a skillset, and an indexer and have the indexer complete one run. We recommend that you begin with a "development" index before moving forward with solution. While the index is in development, use a debug session to inspect the entire structure and contents of the enriched document that will be written into the index. A single run of a debug session works with one specific live document, and will have the content generated by the language model show up in a specific part of the enriched document. Developers can utilize several runs of their debug session, pointing to different documents from their data source to get a reasonable idea of the state of the content produced by their language model (and its relationship to the enriched document structure).

The screenshots below show how developers can inspect both the configuration of a skill and the values produced by the skill after calling the language model.

Example: Inspect the configuration of the GenAI Prompt skill

The screenshot shows the configuration interface for a GenAI Prompt skill named "genai-prompt-skill". The interface includes a toolbar with Run, Stop, Commit changes, Add new skill, Undo, Redo, Save, Refresh, Settings, and Delete buttons. A status bar at the top indicates "Completed". The main area displays a flowchart with a "Document" input node connected to a "genai-pro... Chat Completion" node (Custom). The "genai-pro..." node has "Iterations: 1". To the right, there's a "Skill Settings" panel titled "Skill: genai-prompt-skill" showing "Iterations (1)", "Skill Settings", and "Errors/Warnings (0)". Below it is a table for "Iterations" with one row. The "Path" column shows the skill's path: "InputText" → "\$/document/content", "systemMessage" → "You are an expert AI agent, tasked with understanding and explaining the contents of arbitrary documents, which can have a mix of different language. Your answer will be precise, short and if there are multiple answers, they need to be separated by a comma", and "userMessage" → "What are the different languages in this document?". The "Outputs" column shows "response" with the path "/document/summary". On the far right, there are "Save" and "Cancel" buttons, and a magnifying glass icon.

Example: Inspect the output from the GenAI Prompt skill

The screenshot shows the "Expression evaluator" interface. It features a toolbar with Run, Stop, Commit changes, Add new skill, Undo, Redo, Save, Refresh, Settings, and a search icon. A status bar at the top indicates "Completed". The main area displays a flowchart similar to the previous one, with a "Document" input node and a "genai-pro... Chat Completion" node (Custom). A modal window titled "Expression evaluator" is open, showing the value "1" and the expression "English, Spanish". The "File Preview" tab is selected. To the right, there's a "Skill Settings" panel titled "Path" showing the skill's path: "document/content" → "You are an expert AI agent, tasked with understanding and explaining the contents of arbitrary documents, which can have a mix of different language. Your answer will be precise, short and if there are multiple answers, they need to be separated by a comma" and "What are the different languages in this document?". The "Outputs" column shows "response" with the path "/document/summary". On the far right, there are "Save" and "Cancel" buttons, and a magnifying glass icon.

Use Search Explorer to inspect output

In addition to debug sessions, Azure AI Search also offers the ability to explore multiple documents at once by querying the search index via the Azure portal [search explorer](#). Developers can issue a broad query to retrieve a large number of documents from their search index and can inspect the fields which have their content generated by the GenAI Prompt custom skill. To be able to view the contents of the field, when defining the index schema it

needs to be configured with the "Retrievable" property. For the same document that was inspected via the debug session, the image below shows the full contents of the search document that ends up into the index.

The screenshot shows the Azure AI Search interface. At the top, there are navigation links: Save, Discard, Refresh, Create demo app, Edit JSON, Delete, and Encryption. Below these are metrics: Documents (1), Total storage (6.05 KB), Vector index quota usage (0 Bytes), and Max storage (2 GB). The 'Search explorer' tab is selected, followed by Fields, CORS, Scoring profiles, Semantic configurations, and Vector profiles. On the right, there are 'Query options' and 'View' buttons. A search bar at the bottom has the placeholder 'Search' and a 'Search' button. The main area is titled 'Results' and displays a JSON response from a search query. The JSON output is as follows:

```
1 {  
2   "@odata.context": "https://arjagann-basic-canary.search.windows.net/indexes('genai-prompt-index')/$metadata",  
3   "@odata.count": 1,  
4   "value": [  
5     {  
6       "@search.score": 1,  
7       "id": "aHR0cHM6Ly9hcmpbz2Fubmpma2ZpbGVzLmJsb2IuY29yZS53aw5kb3dzLm5ldC9ha2ovSGVsbG8lMjB3b3JsZC50eHQ1",  
8       "content": "Hello world\r\nHola world\r\n",  
9       "summary": "English, Spanish"  
10    }  
11  ]  
12 }
```

Developers can utilize both of these tools to have a reasonable sampling of the data generated by their language models after it gets ingested into their "development" index. At this point, we recommend developers to work with data SMEs to ensure the sampled data meets quality and accuracy standards. Once desired bars have been met, developers can then transition the indexer to target a "production" index which will be the source of knowledge for their RAG applications.

Content safety filtering and evaluation

Azure AI content safety is an AI service that detects harmful user-generated and AI-generated content in applications and services. Azure AI Foundry has a [robust integration with the Azure AI content safety service](#) for many scenarios across both text and image content. The GenAI Prompt custom skill is flexible enough to allow developers to specify any kinds of prompts and content, since we want developers to tailor the skill configuration to their scenario. But this means that there's a possibility that the content generated by the model could be harmful, biased, explicit, or violent in nature. We highly encourage developers, who configure the GenAI Prompt skill to interact with Azure AI Foundry language models, to set up appropriate content safety filters for the models that they have deployed.

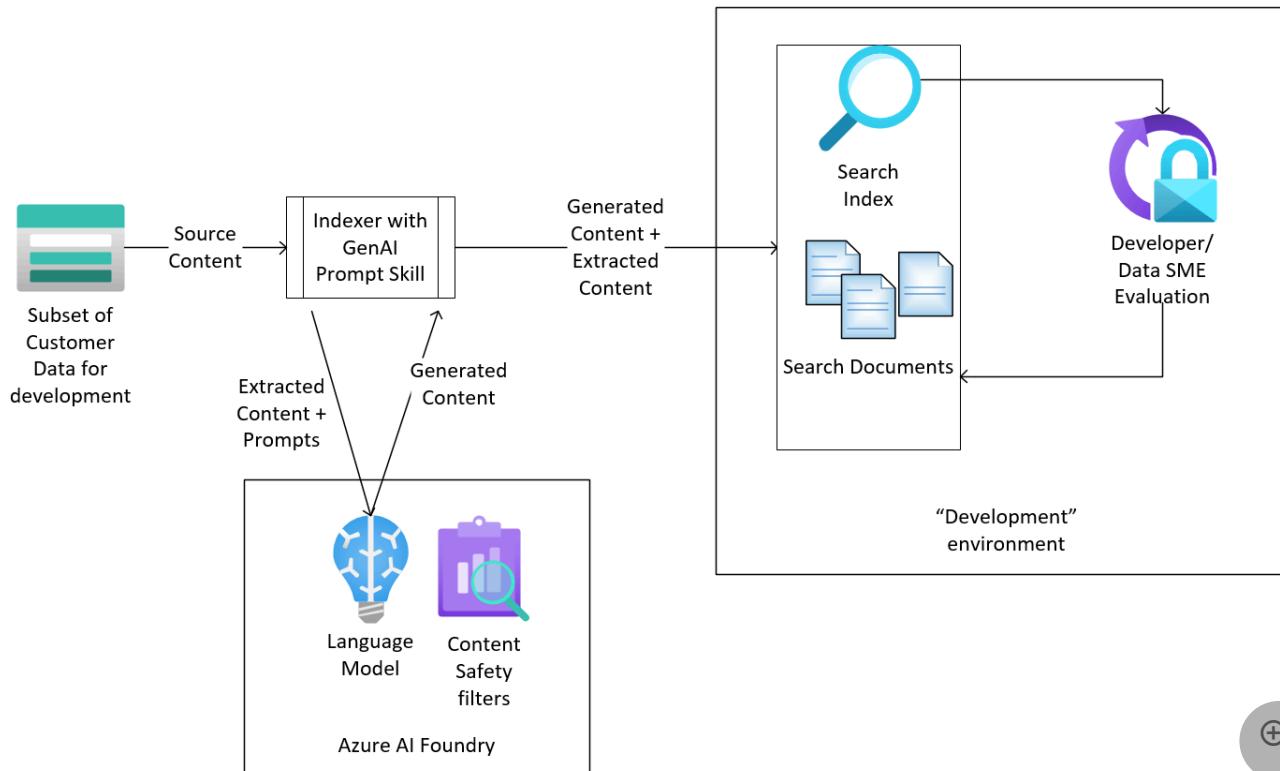
Azure AI Foundry has an [elaborate guide on responsible AI practices](#) and we strongly recommend developers review the contents of the guide and ensure they have implemented

them for the models being used with the GenAI Prompt custom skill. One of the key factors that can determine quality and accuracy of results produced by a language model is a well-crafted system message. Azure AI Foundry also has published a [system message template guide](#) to mitigate against potential RAI harms and guide systems to interact safely with users.

Azure AI Foundry has also published a [detailed guide on evaluation criteria](#) for selecting the models that developers can choose to deploy for their various applications. While the GenAI Prompt skill is flexible enough to work with almost all chat completion models in Azure AI Foundry, depending on the criteria for the end-user, some models might work better than others. Certain chat completion models have advanced image processing capabilities at a high level of accuracy, but come at the expense of cost. Some models might be more prone to attacks via prompts, incorrect information, and so on. Developers who configure the GenAI Prompt custom skill must be aware of these characteristics of the models they choose to utilize for content generation and ingestion into the search index. We strongly recommend developers utilize the "development" index and many sampling to clearly understand the nature of these models as they interact with data before switching it over to production.

Being agile in rolling back or modifying ingestion configuration

The previous two sections stressed the importance for developers to have a "development" environment, where they configure a "development" index with a built-in mechanism for developers/data SMEs to evaluate the contents of the search index.

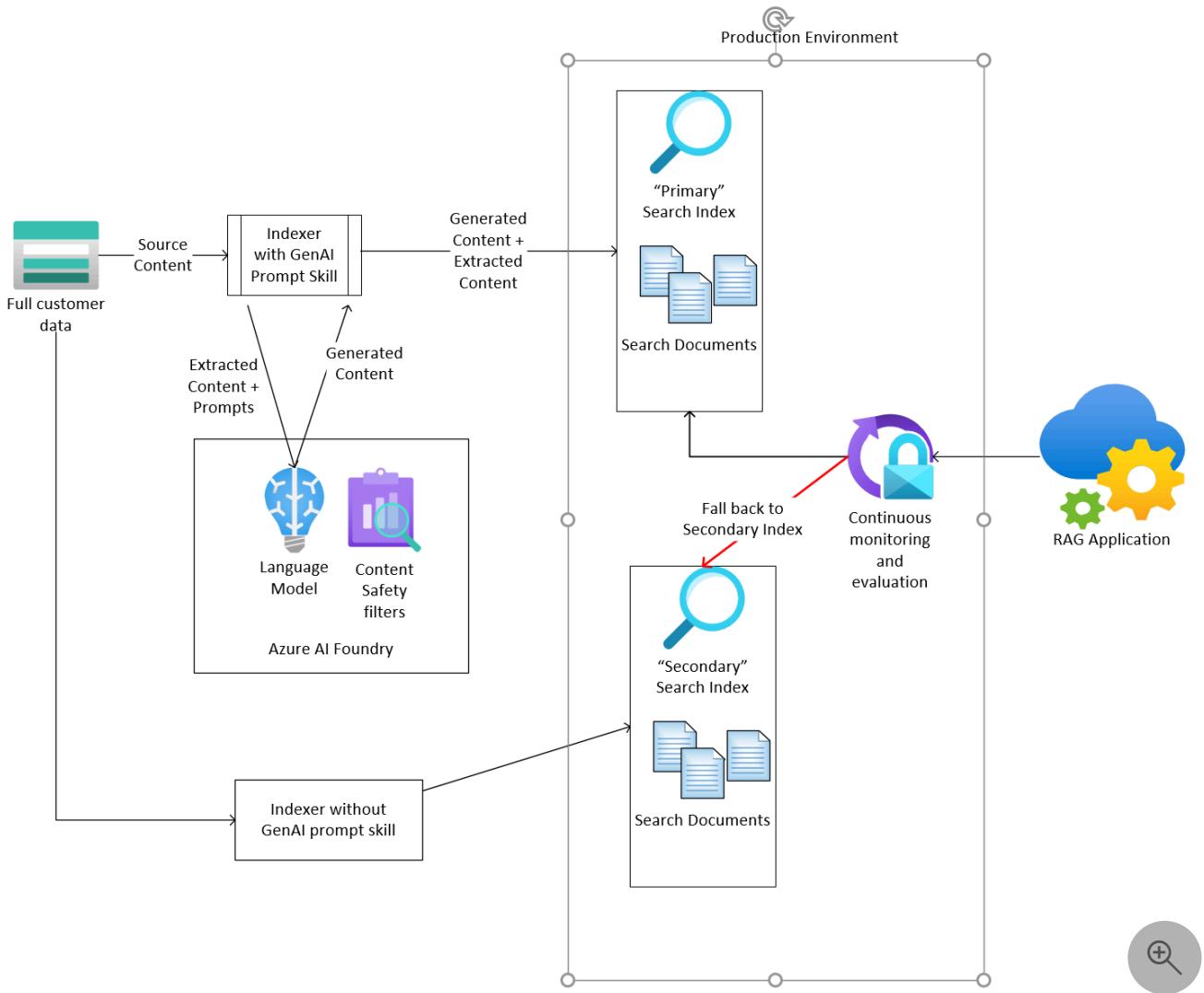


Once the evaluation in the development environment is satisfactory, developers should transition the ingestion process to a production environment, where the indexer operates on the full customer data. However, it's possible for there to be unexpected drops in quality or performance when operating on this data set. It's also possible for the model to be updated without undergoing evaluation in the development environment - both these cases can result in a suboptimal experience for users interacting with RAG applications, and developers need to be agile in detecting and mitigating such conditions. To catch such situations, developers should ensure that they also have a constant monitoring of their "production" index and be ready to modify configurations as needed. The following sections describe some patterns developers could adopt to be responsive to such scenarios.

Primary-Secondary index powering RAG applications

Developers should consider having a primary and a secondary index to power their RAG applications. The primary and secondary indexes would be similar in the configuration of fields - the only difference would be that the primary index will have an extra (searchable and retrievable) field which will contain content generated from the language model through the GenAI Prompt custom skill. Developers should configure their RAG applications such that the AI model being augmented can use either the primary or the secondary index as its knowledge source. The primary index should be preferred, but if the quality of the results produced by the RAG application seems to be adversely impacted, the application should swap to using the secondary index which doesn't have generated content as part of the knowledge source. This can be achieved without needing any code change/redeployment of the RAG app by utilizing the [index alias feature](#) and having the RAG application query the alias, and then swapping the indexes that map to the alias if necessary.

The following diagram illustrates this pattern.



Dropping use of generated field in search queries

A lighter weight alternative to having two copies of the search index, is to ensure that the RAG application can modify the search query issued to Azure AI Search easily. By default when a search query is issued, all searchable fields are scanned, however Azure AI Search allows specifying which fields must be analyzed to produce a set of search results.

Consider an index which has two content fields – a "name" field and a (verbose) "description" field for a sample data set containing hotel information. Let's say the RAG developer/search index admin has configured a third field called "summary", which will hold an AI generated summary of the description via the GenAI Prompt skill. When configuring a RAG application, developers can specify two kinds of search queries - the default search query which analyzes all searchable fields, and another specific query that only looks at those searchable fields whose content isn't generated by a language model. The following is an example of such a specific query, that explicitly omits the "summary" field.

HTTP

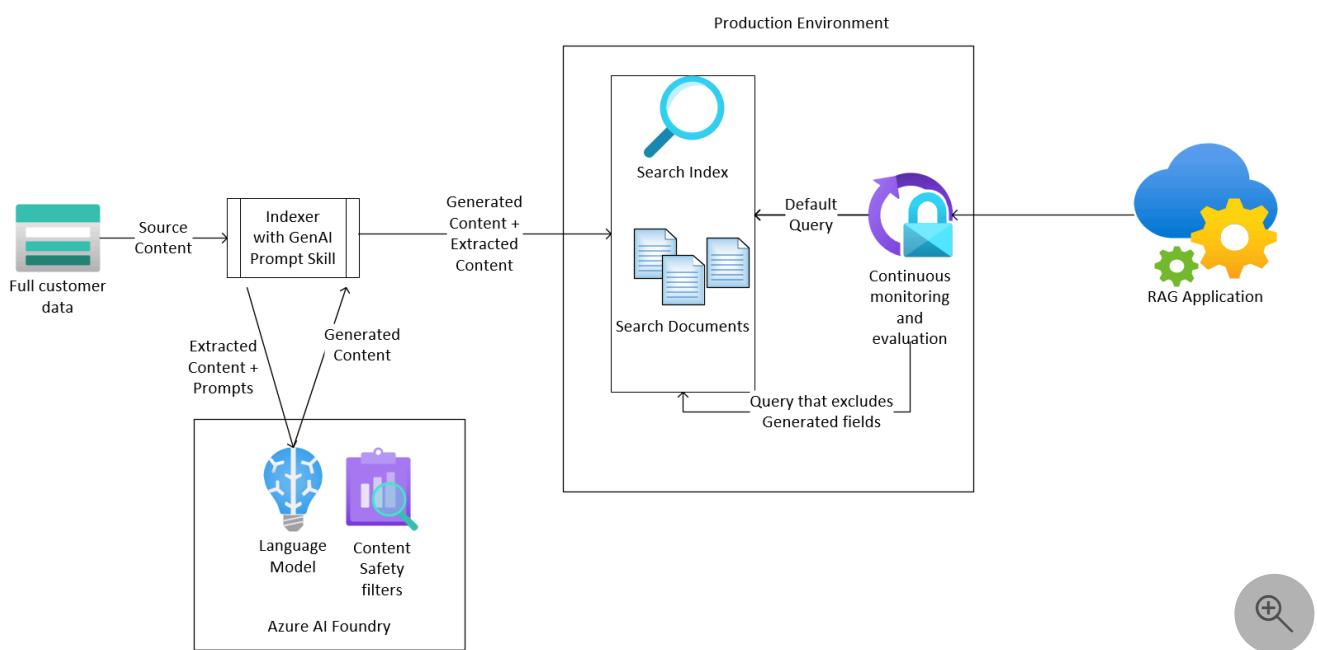
```

POST https://[service-name].search.windows.net/indexes/[index-name]?api-version=[api-version]

{
  "search": "Spacious, air-condition* +\"Ocean view\"",
  "searchFields": "description, name",
  "searchMode": "any",
  "queryType": "full"
}

```

The RAG application can fall back to this specific query (might require a code change/redeployment), if the default query starts to degrade in performance or evaluation metrics, illustrated by the following diagram.



Resetting indexer after modifying skill configuration

Sometimes the system or user message specified to the GenAI Prompt custom skill might be impacted due to new data in the "full" customer data set and tuning these messages might provide a faster alternative than the previous options. However we recommend developers to also [reset and rerun](#) an indexer if the system or user messages are modified for the GenAI Prompt custom skill, to apply this to all documents in the data source. This option can possibly incur additional costs.

Limitations

The risk mitigation strategies and patterns suggested in this document can have limitations based on either scenario specific or data dependent reasons as well as for storage and cost

reasons. It's the responsibility of the RAG developer to ensure that they fully understand any specific risks associated with AI content generation as part of the ingestion pipeline and have appropriate strategies in place to mitigate them.

Human in the loop oversight

Given the scale of data ingestion, it might not be feasible to have a human in the loop oversight for "production" applications. We recommend extensive validation by data SMEs in the "development" environment to minimize the need for this in production. Follow an evaluation process that: (1) uses some internal stakeholders to evaluate results, (2) uses A/B experimentation to roll out this capability to users and (3) incorporates key performance indicators (KPIs) and metrics monitoring when the capability is deployed.

Learn more about responsible AI

- [Azure AI Search transparency note](#)
- [Microsoft AI principles ↗](#)
- [Microsoft responsible AI resources ↗](#)

Learn more about Azure AI Search

- [Introduction to Azure AI Search](#)
- [AI enrichment concepts](#)
- [Retrieval Augmented Generation \(RAG\) in Azure AI Search](#)

Add a custom skill to an Azure AI Search enrichment pipeline

Article • 01/15/2025

An [AI enrichment pipeline](#) can include both [built-in skills](#) and [custom skills](#) that you personally create and publish. Your custom code executes externally from the search service (for example, as an Azure function), but accepts inputs and sends outputs to the skillset just like any other skill. Your data is processed in the [Geo](#) where your model is deployed.

Custom skills might sound complex but can be simple and straightforward in terms of implementation. If you have existing packages that provide pattern matching or classification models, the content you extract from blobs could be passed to these models for processing. Since AI enrichment is Azure-based, your model should be on Azure also. Some common hosting methodologies include using [Azure Functions](#) or [Containers](#).

If you're building a custom skill, this article describes the interface you use to integrate the skill into the pipeline. The primary requirement is the ability to accept inputs and emit outputs in ways that are consumable within the [skillset](#) as a whole. As such, the focus of this article is on the input and output formats that the enrichment pipeline requires.

Benefits of custom skills

Building a custom skill gives you a way to insert transformations unique to your content. For example, you could build custom classification models to differentiate business and financial contracts and documents, or add a speech recognition skill to reach deeper into audio files for relevant content. For a step-by-step example, see [Example: Creating a custom skill for AI enrichment](#).

Set the endpoint and time-out interval

The interface for a custom skill is specified through the [Custom Web API skill](#).

JSON

```
"@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
"description": "This skill has a 230 second time-out",
"uri": "https://[your custom skill uri goes here]",
```

```
"authResourceId": "[for managed identity connections, your app's client ID goes here]",  
"timeout": "PT230S",
```

The URI is the HTTPS endpoint of your function or app. When setting the URI, make sure the URI is secure (HTTPS). If your code is hosted in an Azure function app, the URI should include an [API key in the header or as a URI parameter](#) to authorize the request.

If instead your function or app uses Azure managed identities and Azure roles for authentication and authorization, the custom skill can include an authentication token on the request. The following points describe the requirements for this approach:

- The search service, which sends the request on the indexer's behalf, must be [configured to use a managed identity](#) (either system or user-assigned) so that the caller can be authenticated by Microsoft Entra ID.
- Your function or app must be [configured for Microsoft Entra ID](#).
- Your [custom skill definition](#) must include an `authResourceId` property. This property takes an application (client) ID, in a [supported format](#): `api://<appId>`.

By default, the connection to the endpoint times out if a response isn't returned within a 30-second window (`PT30S`). The indexing pipeline is synchronous and indexing will produce a time-out error if a response isn't received in that time frame. You can increase the interval to a maximum value of 230 seconds by setting the `timeout` parameter (`PT230S`).

Format Web API inputs

The Web API must accept an array of records to be processed. Within each record, provide a property bag as input to your Web API.

Suppose you want to create a basic enricher that identifies the first date mentioned in the text of a contract. In this example, the custom skill accepts a single input "contractText" as the contract text. The skill also has a single output, which is the date of the contract. To make the enricher more interesting, return this "contractDate" in the shape of a multipart complex type.

Your Web API should be ready to receive a batch of input records. Each member of the "values" array represents the input for a particular record. Each record is required to have the following elements:

- A "recordId" member that is the unique identifier for a particular record. When your enricher returns the results, it must provide this "recordId" in order to allow the caller to match the record results to their input.
- A "data" member, which is essentially a bag of input fields for each record.

The resulting Web API request might look like this:

```
JSON

{
  "values": [
    {
      "recordId": "a1",
      "data": {
        "contractText": "This is a contract that was issued on November 3, 2023 and that involves..."
      }
    },
    {
      "recordId": "b5",
      "data": {
        "contractText": "In the City of Seattle, WA on February 5, 2018 there was a decision made..."
      }
    },
    {
      "recordId": "c3",
      "data": {
        "contractText": null
      }
    }
  ]
}
```

In practice, your code can be called with hundreds or thousands of records instead of only the three shown here.

Format Web API outputs

The format of the output is a set of records containing a "recordId", and a property bag. This particular example has only one output, but you could output more than one property. As a best practice, consider returning error and warning messages if a record couldn't be processed.

JSON

```
{  
  "values":  
  [  
    {  
      "recordId": "b5",  
      "data" :  
      {  
        "contractDate": { "day" : 5, "month": 2, "year" : 2018 }  
      }  
    },  
    {  
      "recordId": "a1",  
      "data" : {  
        "contractDate": { "day" : 3, "month": 11, "year" : 2023 }  
      }  
    },  
    {  
      "recordId": "c3",  
      "data" :  
      {  
      },  
      "errors": [ { "message": "contractText field required "} ],  
      "warnings": [ {"message": "Date not found" } ]  
    }  
  ]  
}
```

Add a custom skill to a skillset

When you create a Web API enricher, you can describe HTTP headers and parameters as part of the request. The following snippet shows how request parameters and optional HTTP headers can be included in the skillset definition. Setting an HTTP header is useful if you need to pass configuration settings to your code.

JSON

```
{  
  "skills": [  
    {  
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
      "name": "myCustomSkill",  
      "description": "This skill calls an Azure function, which in turn  
calls TA sentiment",  
      "uri": "https://indexer-e2e-  
webskill.azurewebsites.net/api/DateExtractor?language=en",  
      "context": "/document",  
      "httpHeaders": {  
        "DateExtractor-Api-Key": "foo"  
      }  
    }  
  ]  
}
```

```
        },
        "inputs": [
            {
                "name": "contractText",
                "source": "/document/content"
            }
        ],
        "outputs": [
            {
                "name": "contractDate",
                "targetName": "date"
            }
        ]
    }
}
```

Watch this video

For a video introduction and demo, watch the following demo.

<https://www.youtube-nocookie.com/embed/fHLCE-NZeb4?version=3> ↗

Next steps

This article covered the interface requirements necessary for integrating a custom skill into a skillset. Continue with these links to learn more about custom skills and skillset composition.

- [Power Skills: a repository of custom skills](#) ↗
- [Example: Creating a custom skill for AI enrichment](#)
- [How to define a skillset](#)
- [Create Skillset \(REST\)](#)
- [How to map enriched fields](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗ | [Get help at Microsoft Q&A](#)

Efficiently scale out a custom skill

Article • 01/18/2025

Custom skills are web APIs that implement a specific interface. A custom skill can be implemented on any publicly addressable resource. The most common implementations for custom skills are:

- Azure Functions for custom logic skills
- Azure Web apps for simple containerized AI skills
- Azure Kubernetes service for more complex or larger skills.

Skillset configuration

The following properties on a [custom skill](#) are used for scale. Review the [custom skill interface](#) for an introduction into the inputs and outputs that a custom skill should implement.

1. Set `batchSize` of the custom skill to configure the number of records sent to the skill in a single invocation of the skill.
2. Set the `degreeOfParallelism` to calibrate the number of concurrent requests the indexer makes to your skill.
3. Set `timeout` to a value sufficient for the skill to respond with a valid response.
4. In the `indexer` definition, set `batchSize` to the number of documents that should be read from the data source and enriched concurrently.

Considerations

There's no "one size fits all" set of recommendations. You should plan on testing different configurations to reach an optimum result. Scale up strategies are based on fewer large requests, or many small requests.

- Skill invocation cardinality: make sure you know whether the custom skill executes once for each document (`/document/content`) or multiple times per document (`/document/reviews_text/pages/*`). If it's multiple times per document, stay on the lower side of `batchSize` and `degreeOfParallelism` to reduce churn, and try setting indexer batch size to incrementally higher values for more scale.

- Coordinate custom skill `batchSize` and indexer `batchSize`, and make sure you're not creating bottlenecks. For example, if the indexer batch size is 5, and the skill batch size is 50, you would need 10 indexer batches to fill a custom skill request. Ideally, skill batch size should be less than or equal to indexer batch size.
- For `degreeOfParallelism`, use the average number of requests an indexer batch can generate to guide your decision on how to set this value. If your infrastructure hosting the skill, for example an Azure function, can't support high levels of concurrency, consider dialing down the degrees of parallelism. You can test your configuration with a few documents to validate your understanding of average number of requests.
- Although your object is scale and support of high volumes, testing with a smaller sample of documents helps quantify different stages of execution. For example, you can evaluate the execution time of your skill, relative to the overall time taken to process the subset of documents. This helps you answer the question: does your indexer spend more time building a batch or waiting for a response from your skill?
- Consider the upstream implications of parallelism. If the input to a custom skill is an output from a prior skill, are all the skills in the skillset scaled out effectively to minimize latency?

Error handling in the custom skill

Custom skills should return a success status code HTTP 200 when the skill completes successfully. If one or more records in a batch result in errors, consider returning multi-status code 207. The errors or warnings list for the record should contain the appropriate message.

Any items in a batch that errors will result in the corresponding document failing. If you need the document to succeed, return a warning.

Any status code over 299 is evaluated as an error and all the enrichments are failed resulting in a failed document.

Common error messages

- Could not execute skill because it did not execute within the time limit '00:00:30'. This is likely transient. Please try again later. For custom skills, consider increasing the 'timeout' parameter on your skill in the

`skillset`. Set the timeout parameter on the skill to allow for a longer execution duration.

- `Could not execute skill because Web Api skill response is invalid.` Indicative of the skill not returning a message in the custom skill response format. This could be the result of an uncaught exception in the skill.
- `Could not execute skill because the Web Api request failed.` Most likely caused by authorization errors or exceptions.
- `Could not execute skill.` Commonly the result of the skill response being mapped to an existing property in the document hierarchy.

Testing custom skills

Start by testing your custom skill with a REST API client to validate:

- The skill implements the custom skill interface for requests and responses
- The skill returns valid JSON with the `application/JSON` MIME type
- Returns a valid HTTP status code

Create a [debug session](#) to add your skill to the skillset and make sure it produces a valid enrichment. While a debug session doesn't allow you to tune the performance of the skill, it enables you to ensure that the skill is configured with valid values and returns the expected enriched objects.

Best practices

- While skills can accept and return larger payloads, consider limiting the response to 150 MB or less when returning JSON.
- Consider setting the batch size on the indexer and skill to ensure that each data source batch generates a full payload for your skill.
- For long running tasks, set the timeout to a high enough value to ensure the indexer doesn't error out when processing documents concurrently.
- Optimize the indexer batch size, skill batch size, and skill degrees of parallelism to generate the load pattern your skill expects, fewer large requests or many small requests.

- Monitor custom skills with detailed logs of failures as you can have scenarios where specific requests consistently fail as a result of the data variability.
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Example: Create a custom skill using the Bing Entity Search API

Article • 01/18/2025

In this example, learn how to create a web API custom skill. This skill will accept locations, public figures, and organizations, and return descriptions for them. The example uses an [Azure Function](#) to wrap the [Bing Entity Search API](#) so that it implements the custom skill interface.

Prerequisites

- Read about [custom skill interface](#) article if you aren't familiar with the input/output interface that a custom skill should implement.
- Create a [Bing Search resource](#) through the Azure portal. A free tier is available and sufficient for this example.
- Install [Visual Studio](#) or later.

Create an Azure Function

Although this example uses an Azure Function to host a web API, it isn't required. As long as you meet the [interface requirements for a cognitive skill](#), the approach you take is immaterial. Azure Functions, however, make it easy to create a custom skill.

Create a project

1. In Visual Studio, select **New > Project** from the File menu.
2. Choose **Azure Functions** as the template and select **Next**. Type a name for your project, and select **Create**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other special characters.
3. Select a framework that has long term support.
4. Choose **HTTP Trigger** for the type of function to add to the project.
5. Choose **Function** for the authorization level.
6. Select **Create** to create the function project and HTTP triggered function.

Add code to call the Bing Entity API

Visual Studio creates a project with boilerplate code for the chosen function type. The *FunctionName* attribute on the method sets the name of the function. The *HttpTrigger* attribute specifies that the function is triggered by an HTTP request.

Replace the contents of *Function1.cs* with the following code:

```
C#  
  
using System;  
using System.Collections.Generic;  
using System.IO;  
using System.Linq;  
using System.Net.Http;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;  
using Microsoft.Azure.WebJobs.Extensions.Http;  
using Microsoft.AspNetCore.Http;  
using Microsoft.Extensions.Logging;  
using Newtonsoft.Json;  
  
namespace SampleSkills  
{  
    /// <summary>  
    /// Sample custom skill that wraps the Bing entity search API to connect  
    it with a  
    /// AI enrichment pipeline.  
    /// </summary>  
    public static class BingEntitySearch  
    {  
        #region Credentials  
        // IMPORTANT: Make sure to enter your credential and to verify the  
        API endpoint matches yours.  
        static readonly string bingApiEndpoint =  
"https://api.bing.microsoft.com/v7.0/entities";  
        static readonly string key = "<enter your api key here>";  
        #endregion  
  
        #region Class used to deserialize the request  
        private class InputRecord  
        {  
            public class InputRecordData  
            {  
                public string Name { get; set; }  
            }  
  
            public string RecordId { get; set; }  
            public InputRecordData Data { get; set; }  
        }  
  
        private class WebApiRequest
```

```

{
    public List<InputRecord> Values { get; set; }
}
#endregion

#region Classes used to serialize the response

private class OutputRecord
{
    public class OutputRecordData
    {
        public string Name { get; set; } = "";
        public string Description { get; set; } = "";
        public string Source { get; set; } = "";
        public string SourceUrl { get; set; } = "";
        public string LicenseAttribution { get; set; } = "";
        public string LicenseUrl { get; set; } = "";
    }

    public class OutputRecordMessage
    {
        public string Message { get; set; }
    }

    public string RecordId { get; set; }
    public OutputRecordData Data { get; set; }
    public List<OutputRecordMessage> Errors { get; set; }
    public List<OutputRecordMessage> Warnings { get; set; }
}

private class WebApiResponse
{
    public List<OutputRecord> Values { get; set; }
}
#endregion

#region Classes used to interact with the Bing API
private class BingResponse
{
    public BingEntities Entities { get; set; }
}
private class BingEntities
{
    public BingEntity[] Value { get; set; }
}

private class BingEntity
{
    public class EntityPresentationinfo
    {
        public string[] EntityTypeHints { get; set; }
    }

    public class License
    {

```

```

        public string Url { get; set; }

    }

    public class ContractualRule
    {
        public string _type { get; set; }
        public License License { get; set; }
        public string LicenseNotice { get; set; }
        public string Text { get; set; }
        public string Url { get; set; }
    }

    public ContractualRule[] ContractualRules { get; set; }
    public string Description { get; set; }
    public string Name { get; set; }
    public EntityPresentationinfo EntityPresentationInfo { get; set; }
}

}

#endregion

#region The Azure Function definition

[FunctionName("EntitySearch")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)]
HttpRequest req,
    ILogger log)
{
    log.LogInformation("Entity Search function: C# HTTP trigger
function processed a request.");

    var response = new WebApiResponse
    {
        Values = new List<OutputRecord>()
    };

    string requestBody = new StreamReader(req.Body).ReadToEnd();
    var data = JsonConvert.DeserializeObject<WebApiRequest>
(requestBody);

    // Do some schema validation
    if (data == null)
    {
        return new BadRequestObjectResult("The request schema does
not match expected schema.");
    }
    if (data.Values == null)
    {
        return new BadRequestObjectResult("The request schema does
not match expected schema. Could not find values array.");
    }

    // Calculate the response for each value.
    foreach (var record in data.Values)
    {

```

```

        if (record == null || record.RecordId == null) continue;

        OutputRecord responseRecord = new OutputRecord
        {
            RecordId = record.RecordId
        };

        try
        {
            responseRecord.Data =
GetEntityMetadata(record.Data.Name).Result;
        }
        catch (Exception e)
        {
            // Something bad happened, log the issue.
            var error = new OutputRecord.OutputRecordMessage
            {
                Message = e.Message
            };

            responseRecord.Errors = new
List<OutputRecord.OutputRecordMessage>
            {
                error
            };
        }
        finally
        {
            response.Values.Add(responseRecord);
        }
    }

    return (ActionResult)new OkObjectResult(response);
}

#endregion

#region Methods to call the Bing API
/// <summary>
/// Gets metadata for a particular entity based on its name using
Bing Entity Search
/// </summary>
/// <param name="entityName">The name of the entity to extract data
for.</param>
/// <returns>Asynchronous task that returns entity data. </returns>
private async static Task<OutputRecord.OutputRecordData>
GetEntityMetadata(string entityName)
{
    var uri = bingApiEndpoint + "?q=" + entityName + "&mkt=en-
us&count=10&offset=0&safesearch=Moderate";
    var result = new OutputRecord.OutputRecordData();

    using (var client = new HttpClient())
    using (var request = new HttpRequestMessage {
        Method = HttpMethod.Get,

```



```

rule.LicenseNotice;
rootObject.LicenseUrl =
rule.License.Url;
break;
case "ContractualRules/LinkAttribution":
rootObject.Source = rule.Text;
rootObject.SourceUrl = rule.Url;
break;
}
}
}

return rootObject;
}
}

return new OutputRecord.OutputRecordData();
}
#endregion
}
}

```

Make sure to enter your own *key* value in the `key` constant based on the key you got when signing up for the Bing Entity search API.

Test the function from Visual Studio

Press **F5** to run the program and test function behaviors. In this case, we'll use the function below to look up two entities. Use a REST client to issue a call like the one shown below:

HTTP

`POST https://localhost:7071/api/EntitySearch`

Request body

JSON

```
{
  "values": [
    {
      "recordId": "e1",
      "data": {
        "name": "Pablo Picasso"
      }
    },
  ],
}
```

```
        {
            "recordId": "e2",
            "data":
            {
                "name": "Microsoft"
            }
        }
    ]
}
```

Response

You should see a response similar to the following example:

JSON

```
{
    "values": [
        {
            "recordId": "e1",
            "data": {
                "name": "Pablo Picasso",
                "description": "Pablo Ruiz Picasso was a Spanish painter
[...]",
                "source": "Wikipedia",
                "sourceUrl": "http://en.wikipedia.org/wiki/Pablo_Picasso",
                "licenseAttribution": "Text under CC-BY-SA license",
                "licenseUrl": "http://creativecommons.org/licenses/by-
sa/3.0/"
            },
            "errors": null,
            "warnings": null
        },
        ...
    ]
}
```

Publish the function to Azure

When you're satisfied with the function behavior, you can publish it.

1. In **Solution Explorer**, right-click the project and select **Publish**. Choose **Create New** > **Publish**.
2. If you haven't already connected Visual Studio to your Azure account, select **Add an account....**

3. Follow the on-screen prompts. You're asked to specify a unique name for your app service, the Azure subscription, the resource group, the hosting plan, and the storage account you want to use. You can create a new resource group, a new hosting plan, and a storage account if you don't already have these. When finished, select **Create**
4. After the deployment is complete, notice the Site URL. It is the address of your function app in Azure.
5. In the [Azure portal](#), navigate to the Resource Group, and look for the **EntitySearch** Function you published. Under the **Manage** section, you should see Host Keys. Select the **Copy** icon for the *default* host key.

Test the function in Azure

Now that you have the default host key, test your function as follows:

```
HTTP  
  
POST https://[your-entity-search-app-name].azurewebsites.net/api/EntitySearch?code=[enter default host key here]
```

Request Body

```
JSON  
  
{  
  "values": [  
    {  
      "recordId": "e1",  
      "data":  
        {  
          "name": "Pablo Picasso"  
        }  
    },  
    {  
      "recordId": "e2",  
      "data":  
        {  
          "name": "Microsoft"  
        }  
    }  
  ]  
}
```

This example should produce the same result you saw previously when running the function in the local environment.

Connect to your pipeline

Now that you have a new custom skill, you can add it to your skillset. The example below shows you how to call the skill to add descriptions to organizations in the document (this could be extended to also work on locations and people). Replace [your-entity-search-app-name] with the name of your app.

```
JSON

{
  "skills": [
    "[... your existing skills remain here]",
    {
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
      "description": "Our new Bing entity search custom skill",
      "uri": "https://[your-entity-search-app-
name].azurewebsites.net/api/EntitySearch?code=[enter default host key
here]",
      "context": "/document/merged_content/organizations/*",
      "inputs": [
        {
          "name": "name",
          "source": "/document/merged_content/organizations/*"
        }
      ],
      "outputs": [
        {
          "name": "description",
          "targetName": "description"
        }
      ]
    }
  ]
}
```

Here, we're counting on the built-in [entity recognition skill](#) to be present in the skillset and to have enriched the document with the list of organizations. For reference, here's an entity extraction skill configuration that would be sufficient in generating the data we need:

```
JSON

{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "name": "#1",
```

```
"description": "Organization name extraction",
"context": "/document/merged_content",
"categories": [ "Organization" ],
"defaultLanguageCode": "en",
"inputs": [
  {
    "name": "text",
    "source": "/document/merged_content"
  },
  {
    "name": "languageCode",
    "source": "/document/language"
  }
],
"outputs": [
  {
    "name": "organizations",
    "targetName": "organizations"
  }
]
},
```

Next steps

Congratulations! You've created your first custom skill. Now you can follow the same pattern to add your own custom functionality. Click the following links to learn more.

- [Power Skills: a repository of custom skills ↗](#)
- [Add a custom skill to an AI enrichment pipeline](#)
- [How to define a skillset](#)
- [Create Skillset \(REST\)](#)
- [How to map enriched fields](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create an index for agentic retrieval in Azure AI Search

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In Azure AI Search, agentic retrieval uses context and user questions to generate a range of subqueries that can execute against your content in a [knowledge source](#). A knowledge source can point to indexed content on Azure AI Search, or remote content that's retrieved using the APIs that are native to the provider. When indexes are used in agentic retrieval, they are either:

- An *existing index* containing searchable content. You can make an existing index available to agentic retrieval through a [search index knowledge source](#) definition.
- A *generated index* created from an indexed [knowledge source](#). Indexed knowledge sources create a copy of an external data source inside a search index using a full indexer pipeline (data source, skillset, indexer, and index) for agentic retrieval. Multiple knowledge sources can generate an indexer pipeline that results in a searchable index. These include:
 - [Azure blobs](#)
 - [Microsoft OneLake](#)
 - [SharePoint \(Indexed\)](#)

A generated index is based on a template that meets all of the criteria for knowledge bases and agentic retrieval.

This article explains which index elements affect agentic retrieval query logic. None of the required elements are new or specific to agentic retrieval, which means you can use an existing index if it meets the criteria identified in this article, even if it was created using earlier API versions.

! Note

The following knowledge sources access external sources directly and don't require a search index: [Web Knowledge Source \(Bing\)](#) and [SharePoint \(Remote\)](#).

Criteria for agentic retrieval

An index that's used in agentic retrieval must have these elements:

- String fields attributed as `searchable` and `retrievable`.
- A [semantic configuration](#), with a `defaultSemanticConfiguration` or a semantic configuration override in the knowledge source.

It should also have fields that can be used for citations, such as document or file name, page or chapter name, or at least a chunk ID.

It should have [vector fields and a vectorizer](#) if you want to include text-to-vector query conversion in the pipeline.

Optionally, the following index elements increase your opportunities for optimization:

- `scoringProfile` with a `defaultScoringProfile`, for boosting relevance
- `synonymMaps` for terminology or jargon
- `analyzers` for linguistics rules or patterns (like whitespace preservation, or special characters)

Example index definition

Here's an example index that works for agentic retrieval. It meets the criteria for required elements. It includes vector fields as a best practice.

JSON

```
{  
  "name": "earth_at_night",  
  "description": "Contains images and descriptions of our planet in darkness as  
captured from space by Earth-observing satellites and astronauts on the  
International Space Station over the past 25 years.",  
  "fields": [  
    {  
      "name": "id", "type": "Edm.String",  
      "searchable": true, "retrievable": true, "filterable": true, "sortable": true,  
      "facetable": true,  
      "key": true,  
      "stored": true,  
      "synonymMaps": []  
    },  
    {  
      "name": "page_chunk", "type": "Edm.String",  
      "searchable": true, "retrievable": true, "filterable": false, "sortable":  
false, "facetable": false,  
      "analyzer": "en.microsoft",  
      "vector": true  
    }  
  ]  
}
```

```
        "stored": true,
        "synonymMaps": []
    },
{
    "name": "page_chunk_vector_text_3_large", "type": "Collection(Edm.Single)",
    "searchable": true, "retrievable": false, "filterable": false, "sortable": false, "facetable": false,
    "dimensions": 3072,
    "vectorSearchProfile": "hnsw_text_3_large",
    "stored": false,
    "synonymMaps": []
},
{
    "name": "page_number", "type": "Edm.Int32",
    "searchable": false, "retrievable": true, "filterable": true, "sortable": true, "facetable": true,
    "stored": true,
    "synonymMaps": []
},
{
    "name": "chapter_number", "type": "Edm.Int32",
    "searchable": false, "retrievable": true, "filterable": true, "sortable": true, "facetable": true,
    "stored": true,
    "synonymMaps": []
},
],
"semantic": {
    "defaultConfiguration": "semantic_config",
    "configurations": [
        {
            "name": "semantic_config",
            "flightingOptIn": false,
            "prioritizedFields": {
                "prioritizedContentFields": [
                    {
                        "fieldName": "page_chunk"
                    }
                ],
                "prioritizedKeywordsFields": []
            }
        }
    ]
},
"vectorSearch": {
    "algorithms": [
        {
            "name": "alg",
            "kind": "hnsw",
            "hnswParameters": {
                "metric": "cosine",
                "m": 4,
                "efConstruction": 400,
                "efSearch": 500
            }
        }
    ]
}
```

```

        }
    ],
    "profiles": [
        {
            "name": "hnsw_text_3_large",
            "algorithm": "alg",
            "vectorizer": "azure_openai_text_3_large"
        }
    ],
    "vectorizers": [
        {
            "name": "azure_openai_text_3_large",
            "kind": "azureOpenAI",
            "azureOpenAIParameters": {
                "resourceUri": "https://YOUR-AOAI-RESOURCE.openai.azure.com",
                "deploymentId": "text-embedding-3-large",
                "apiKey": "<redacted>",
                "modelName": "text-embedding-3-large"
            }
        }
    ],
    "compressions": []
}
}

```

Key points:

A well-designed index that's used for generative AI or retrieval augmented generation (RAG) structure has these components:

- A description that an LLM or agent can use to determine whether an index should be used or skipped.
- Chunks of human readable text that can be passed as input tokens to an LLM for answer formulation.
- A semantic ranker configuration because agentic retrieval uses level 2 (L2) semantic ranking to identify the most relevant chunks.
- Optionally, vector-equivalent versions of the human readable chunks of text for complementary vector search.

Chunked text is important because LLMs consume and emit tokenized strings of human readable plain text content. For this reason, you want `searchable` fields that provide plain text strings, and are `retrievable` in the response. In Azure AI Search, chunked text can be created using [built-in or third-party solutions](#).

A built-in assumption for chunked content is that the original source documents have large amounts of verbose content. If your source content is structured data, such as a product

database, then your index should forego chunking and instead include fields that map to the original data source (for example, a product name, category, description, and so forth). Attribution of `searchable` and `retrievable` applies to structured data as well. Searchable makes the content in-scope for queries, and retrievable adds it to the search results (grounding data).

Vector content can be useful because it adds *similarity search* to information retrieval. At query time, when vector fields are present in the index, the agentic retrieval engine executes a vector query in parallel to the text query. Because vector queries look for similar content rather than matching words, a vector query can find a highly relevant result that a text query might miss. Adding vectors can enhance and improve the quality of your grounding data, but aren't otherwise strictly required. Azure AI Search has a [built-in approach for vectorization](#).

Vector fields are used only for query execution on Azure AI Search. You don't need the vector in results because it isn't human or LLM readable. We recommend that you set `retrievable` and `stored` to false to minimize space requirements. For more information, see [Optimize vector storage and processing](#).

If you use vectors, having a `vectorizer` defined in the vector search configuration is critical. It determines whether your vector field is used during query execution. The vectorizer encodes string subqueries into vectors at query time for similarity search over the vectors. The vectorizer must be the same embedding model used to create the vectors in the index.

By default, all `searchable` fields are included in query execution, and all `retrievable` fields are returned in results. You can choose which fields to use for each action in the [search index knowledge source definition](#).

Add a description

An index `description` field is a user-defined string that you can use to provide guidance to LLMs and Model Context Protocol (MCP) servers when deciding to use a specific index for a query. This human-readable text is invaluable when a system must access several indexes and make a decision based on the description.

An index description is a schema update, and you can add it without having to rebuild the entire index.

- String length is 4,000 characters maximum.
- Content must be human-readable, in Unicode. Your use case should determine which language to use (for example, English or another language).

Add a semantic configuration

The index must have at least one semantic configuration. The semantic configuration must have:

- A named configuration.
- A `prioritizedContentFields` set to at least one string field that is both `searchable` and `retrievable`.

There are two ways to specify a semantic configuration by name. If index has `defaultSemanticConfiguration` set to a named configuration, retrieval uses it. Alternatively, you can specify the semantic configuration inside the [search index knowledge source](#).

Within the configuration, `prioritizedContentFields` is required. Title and keywords are optional. For chunked content, you might not have either. However, if you add [entity recognition](#) or [key phrase extraction](#), you might have some keywords associated with each chunk that can be useful in search scenarios, perhaps in a scoring profile.

Here's an example of a semantic configuration that works for agentic retrieval:

JSON

```
"semantic":{  
    "defaultConfiguration":"semantic_config",  
    "configurations": [  
        {  
            "name": "semantic_config",  
            "flightingOptIn": false,  
            "prioritizedFields": {  
                "prioritizedFields": {  
                    "titleField": {  
                        "fieldName": ""  
                    },  
                    "prioritizedContentFields": [  
                        {  
                            "fieldName": "page_chunk"  
                        }  
                    ],  
                    "prioritizedKeywordsFields": [  
                        {  
                            "fieldName": "Category"  
                        },  
                        {  
                            "fieldName": "Tags"  
                        },  
                        {  
                            "fieldName": "Location"  
                        }  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        }
    }
]
}
```

⚠ Note

The response provides `title`, `terms`, and `content`, which map to the prioritized fields in this configuration.

Add a vectorizer

If you have vector fields in the index, the query plan includes them if they're `searchable` and have a `vectorizer` assignment.

A [vectorizer](#) specifies an embedding model that provides text-to-vector conversions at query time. It must point to the same embedding model used to encode the vector content in your index. You can use any embedding model supported by Azure AI Search. Vectorizers are specified on vector fields by way of a *vector profile*.

Recall the **vector field definition** in the index example. Attributes on a vector field include dimensions or the number of embeddings generated by the model, and the profile.

JSON

```
{
  "name": "page_chunk_text_3_large", "type": "Collection(Edm.Single)",
  "searchable": true, "retrievable": false, "filterable": false, "sortable": false,
  "facetable": false,
  "dimensions": 3072,
  "vectorSearchProfile": "hnsw_text_3_large",
  "stored": false,
  "synonymMaps": []
}
```

Vector profiles are configurations of vectorizers, algorithms, and compression techniques. Each vector field can only use one profile, but your index can have many in case you want unique profiles for every vector field.

Querying vectors and calling a vectorizer adds latency to the overall request, but if you want similarity search it might be worth the trade-off.

Here's an example of a vectorizer that works for agentic retrieval, as it appears in a `vectorSearch` configuration. There's nothing in the vectorizer definition that needs to be

changed to work with agentic retrieval.

JSON

```
"vectorSearch": {  
    "algorithms": [  
        {  
            "name": "alg",  
            "kind": "hnsw",  
            "hnswParameters": {  
                "metric": "cosine",  
                "m": 4,  
                "efConstruction": 400,  
                "efSearch": 500  
            }  
        }  
    ],  
    "profiles": [  
        {  
            "name": "hnsw_text_3_large",  
            "algorithm": "alg",  
            "vectorizer": "azure_openai_text_3_large"  
        }  
    ],  
    "vectorizers": [  
        {  
            "name": "azure_openai_text_3_large",  
            "kind": "azureOpenAI",  
            "azureOpenAIParameters": {  
                "resourceUri": "https://YOUR-AOAI-RESOURCE.openai.azure.com",  
                "deploymentId": "text-embedding-3-large",  
                "apiKey": "<redacted>",  
                "modelName": "text-embedding-3-large"  
            }  
        }  
    ],  
    "compressions": []  
}
```

Add a scoring profile

Scoring profiles are criteria for relevance boosting. They're applied to non-vector fields (text and numbers) and are evaluated during query execution, although the precise behavior depends on the API version used to create the index.

A scoring profile is more likely to add value to your solution if your index is based on structured data. Structured data is indexed into multiple discrete fields, which means your scoring profile can have criteria that target the content or characteristics of a specific field.

If you create the index using 2025-05-01-preview or later, the scoring profile executes last. If the index is created using an earlier API version, scoring profiles are evaluated before semantic reranking. Because agentic retrieval is available in newer preview APIs, the scoring profile executes last. The actual order of semantically ranked results is determined by the [rankingOrder](#) property in the index, which is either set to `boostedRerankerScore` (a scoring profile was applied) or `rerankerScore` (no scoring profile).

You can use any scoring profile that makes sense for your index. Here's an example of one that boosts the search score of a match if the match is found in a specific field. Fields are weighted by boosting multipliers. For example if a match was found in the "Category" field, the boosted score is multiplied by 5.

JSON

```
"scoringProfiles": [
  {
    "name": "boostSearchTerms",
    "text": {
      "weights": {
        "Location": 2,
        "Category": 5
      }
    }
  }
]
```

Add analyzers

[Analyzers](#) apply to text fields and can be language analyzers or custom analyzers that control tokenization in the index, such as preserving special characters or whitespace.

Analyzers are defined within a search index and assigned to fields. The [fields collection example](#) includes an analyzer reference on the text chunks. In this example, the default analyzer (standard Lucene) is replaced with a Microsoft language analyzer for the English language.

JSON

```
{
  "name": "page_chunk", "type": "Edm.String",
  "searchable": true, "retrievable": true, "filterable": false, "sortable": false,
  "facetable": false,
  "analyzer": "en.microsoft",
  "stored": true,
  "synonymMaps": []
}
```

Add a synonym map

Synonym maps expand queries by adding synonyms for named terms. For example, you might have scientific or medical terms for common terms.

Synonym maps are defined as a top-level resource on a search index and assigned to fields. The [fields collection example](#) doesn't include a synonym map, but if you had variant spellings of country names in synonym map, here's what an example might look like if it was assigned to a hypothetical "locations" field.

JSON

```
{  
  "name": "locations",  
  "type": "Edm.String",  
  "searchable": true,  
  "synonymMaps": [ "country-synonyms" ]  
}
```

Add your index to a knowledge source

If you have a standalone index that already exists, and it isn't generated by a knowledge source, create the following objects:

- A [searchIndex knowledge source](#) to encapsulate your indexed content.
- A [knowledge base](#) that represents one or more knowledge sources and other instructions for knowledge retrieval.

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

What is a knowledge source?

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

A knowledge source specifies the content used for agentic retrieval. It encapsulates a search index which may be populated by an external data source, or a direct connection to a remote source such as Bing or Sharepoint that is queried directly. A knowledge source is a required definition in a knowledge base.

- Create a knowledge source as a top-level resource on your search service. Each knowledge source points to exactly one data structure, either a search index that [meets the criteria for agentic retrieval](#) or a supported external resource.
- Reference one or more knowledge sources in a knowledge base. In an agentic retrieval pipeline, it's possible to query against multiple knowledge sources in a single request. Subqueries are generated for each knowledge source. Top results are returned in the retrieval response.
- For certain knowledge sources, you can use a knowledge source definition to generate a full indexer pipeline (data source, skillset, indexer, and index) that works for agentic retrieval. Instead of creating multiple objects manually, information in the knowledge source is used to generate all objects, including a populated, chunked, and searchable index.

Make sure you have at least one knowledge source before creating a knowledge base. The full specification of a knowledge source and a knowledge base is in the [preview REST API reference](#).

Working with a knowledge source

- Creation path: first create a knowledge source, then create a knowledge base.
- Deletion path: update or delete knowledge bases to remove references to a knowledge source, and then delete the knowledge source last.
- A knowledge source, its index, and the knowledge base must all exist on the same search service. External content is either accessed over the public internet (Bing) or in a Microsoft

tenant (remote SharePoint).

Supported knowledge sources

Here are the knowledge sources you can create in this preview:

- "[searchIndex](#)" API wraps an existing index.
- "[azureBlob](#)" API generates an indexer pipeline that pulls from a blob container.
- "[indexedOneLake](#)" API generates an indexer pipeline that pulls from a lakehouse.
- "[indexedSharePoint](#)" API generates an indexer pipeline that pulls from a SharePoint site.
- "[remoteSharePoint](#)" API retrieves content directly from SharePoint.
- "[webParameters](#)" API retrieves real-time grounding data from Microsoft Bing.

Creating knowledge sources

You must have [Search Service Contributor](#) permissions to create objects on a search service. You also need [Search Index Data Contributor](#) permissions to load an index if you're using a knowledge source that creates an indexer pipeline. Alternatively, you can [use an API admin key](#) instead of roles.

You can use the REST API or an Azure SDK preview package to create a knowledge source. Azure portal support is available for select knowledge sources. The following links provide instructions for creating a knowledge source:

- [How to create a search index knowledge source \(wraps an existing index\)](#)
- [How to create a blob knowledge source \(generates an indexer pipeline\)](#)
- [How to create a OneLake knowledge source \(generates an indexer pipeline\)](#)
- [How to create a SharePoint \(indexed\) knowledge source \(generates an indexer pipeline\)](#)
- [How to create a SharePoint \(remote\) knowledge source \(queries SharePoint directly\)](#)
- [How to create a Web Knowledge Source resource \(connects to Bing's public endpoint\)](#)

After the knowledge source is created, you can reference it in a knowledge base.

Using knowledge sources

Properties on the [knowledge base](#) determine which knowledge sources are used.

- "[knowledgeSources](#)" REST array specifies the knowledge sources available to the knowledge base.
- "[retrievalReasoningEffort](#)" REST properties determine the amount of effort put into a retrieval. For more information, see [Set the retrieval reasoning effort](#).

- "outputMode" REST affects query output and what goes in the response.

The knowledge base uses the [retrieve action](#) to send queries to the index specified in the knowledge source. In the retrieve action, some knowledge base and source defaults can be overridden at retrieval time.

Use multiple knowledge sources simultaneously

When you have multiple knowledge sources, set the following properties to bias query planning to a specific knowledge source.

- Setting `alwaysQuerySource` forces query planning to always include the knowledge source.
- Setting `retrievalInstructions` provides guidance that includes or excludes a knowledge source.

Retrieval instructions are sent as a user-defined prompt to the large language model (LLM) used for query planning. This prompt is helpful when you have multiple knowledge sources and want to provide guidance on when to use each one. For example, if you have separate indexes for product information, job postings, and technical support, the retrieval instructions might say "use the jobs index only if the question is about a job application."

The `alwaysQuerySource` property overrides `retrievalInstructions`. Set `alwaysQuerySource` to false when providing retrieval instructions.

Use a retrieval reasoning effort to control LLM usage

Not all solutions benefit from LLM query planning and execution. If simplicity and speed outweigh the benefits the LLM query planning and context engineering provide, you can omit the LLM from your pipeline.

The retrieval reasoning effort determines the level of processing that goes into a retrieval action. For more information, see [Set the retrieval reasoning effort](#).

! Note

If you used `attemptFastPath` in the previous preview, that approach is now replaced with `retrievalReasoningEffort` set to `minimal`.

Create a blob knowledge source from Azure Blob Storage and ADLS Gen2

!Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Use a *blob knowledge source* to index and query Azure blob content in an agentic retrieval pipeline. [Knowledge sources](#) are created independently, referenced in a [knowledge base](#), and used as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

Unlike a [search index knowledge source](#), which specifies an existing and qualified index, a blob knowledge source specifies an external data source, models, and properties to automatically generate the following Azure AI Search objects:

- A data source that represents a blob container.
- A skillset that chunks and optionally vectorizes multimodal content from the container.
- An index that stores enriched content and meets the criteria for agentic retrieval.
- An indexer that uses the previous objects to drive the indexing and enrichment pipeline.

!Note

If user access is specified at the document (blob) level in Azure Storage, a knowledge source can carry permission metadata forward to indexed content in Azure AI Search. For more information, see [ADLS Gen2 permission metadata](#) or [Blob RBAC scopes](#).

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#).
- An [Azure Blob Storage](#) or [Azure Data Lake Storage \(ADLS\) Gen2](#) account.
- A blob container with [supported content types](#) for text content. For optional image verbalization, the supported content type depends on whether your chat completion model can analyze and describe the image file.

- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).

! Note

Although you can use the Azure portal to create blob knowledge sources, the portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}
```

```
response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a blob knowledge source.

JSON

```
{
  "name": "my-blob-ks",
  "kind": "azureBlob",
  "description": "A sample blob knowledge source.",
  "encryptionKey": null,
  "azureBlobParameters": {
    "connectionString": "<REDACTED>",
    "containerName": "blobcontainer",
    "folderPath": null,
    "isADLSGen2": false,
    "ingestionParameters": {
      "disableImageVerbalization": false,
      "ingestionPermissionOptions": [],
      "contentExtractionMode": "standard",
      "identity": null,
      "embeddingModel": {
        "kind": "azureOpenAI",
        "azureOpenAIParameters": {
          "resourceUri": "<REDACTED>",
          "deploymentId": "text-embedding-3-large",
          "apiKey": "<REDACTED>",
          "modelName": "text-embedding-3-large",
          "authIdentity": null
        }
      },
      "chatCompletionModel": {
        "kind": "azureOpenAI",
        "azureOpenAIParameters": {
          "resourceUri": "<REDACTED>",
          "deploymentId": "gpt-5-mini",
          "apiKey": "<REDACTED>",
          "modelName": "gpt-5-mini",
          "authIdentity": null
        }
      },
      "ingestionSchedule": null,
      "assetStore": null,
      "aiServices": {
        "uri": "<REDACTED>",
        "apiKey": "<REDACTED>"
      }
    },
    "createdResources": {
      "datasource": "my-blob-ks-datasource",
      "indexer": "my-blob-ks-indexer",
      "skillset": "my-blob-ks-skillset",
      "index": "my-blob-ks-index"
    }
  }
}
```

```
    }  
}  
}
```

! Note

Sensitive information is redacted. The generated resources appear at the end of the response.

Create a knowledge source

Run the following code to create a blob knowledge source.

Python

```
# Create a blob knowledge source  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient  
from azure.search.documents.indexes.models import AzureBlobKnowledgeSource,  
AzureBlobKnowledgeSourceParameters, KnowledgeBaseAzureOpenAIModel,  
AzureOpenAIVectorizerParameters, KnowledgeSourceAzureOpenAIVectorizer,  
KnowledgeSourceContentExtractionMode, KnowledgeSourceIngestionParameters  
  
index_client = SearchIndexClient(endpoint = "search_url", credential =  
AzureKeyCredential("api_key"))  
  
knowledge_source = AzureBlobKnowledgeSource(  
    name = "my-blob-ks",  
    description = "This knowledge source pulls from a blob storage container.",  
    encryption_key = None,  
    azure_blob_parameters = AzureBlobKnowledgeSourceParameters(  
        connection_string = "blob_connection_string",  
        container_name = "blob_container_name",  
        folder_path = None,  
        is_adls_gen2 = False,  
        ingestion_parameters = KnowledgeSourceIngestionParameters(  
            identity = None,  
            disable_image_verbalization = False,  
            chat_completion_model = KnowledgeBaseAzureOpenAIModel(  
                azure_open_ai_parameters = AzureOpenAIVectorizerParameters(  
                    # TRIMMED FOR BREVITY  
                )  
,  
            embedding_model = KnowledgeSourceAzureOpenAIVectorizer(  
                azure_open_ai_parameters=AzureOpenAIVectorizerParameters(  
                    # TRIMMED FOR BREVITY  
                )  
,  
            content_extraction_mode = KnowledgeSourceContentExtractionMode.MINIMAL,
```

```

        ingestion_schedule = None,
        ingestion_permission_options = None
    )
)
)

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated
successfully.")

```

Source-specific properties

You can pass the following properties to create a blob knowledge source.

[] Expand table

Name	Description	Type	Editable	Required
<code>name</code>	The name of the knowledge source, which must be unique within the knowledge sources collection and follow the naming guidelines for objects in Azure AI Search.	String	No	Yes
<code>description</code>	A description of the knowledge source.	String	Yes	No
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in both the knowledge source and the generated objects.	Object	Yes	No
<code>azure_blob_parameters</code>	Parameters specific to blob knowledge sources: <code>connection_string</code> , <code>container_name</code> , <code>folder_path</code> , and <code>is_adls_gen2</code> .	Object		No
<code>connection_string</code>	A key-based connection string or, if you're using a managed identity, the resource ID.	String	No	Yes
<code>container_name</code>	The name of the blob storage container.	String	No	Yes
<code>folder_path</code>	A folder within the container.	String	No	No
<code>is_adls_gen2</code>	The default is <code>False</code> . Set to <code>True</code> if you're using an ADLS Gen2 storage account.	Boolean	No	No

`ingestionParameters` properties

For indexed knowledge sources only, you can pass the following `ingestionParameters` properties to control how content is ingested and processed.

[Expand table](#)

Name	Description	Type	Editable	Required
<code>identity</code>	A managed identity to use in the generated indexer.	Object	Yes	No
<code>disable_image_verbalization</code>	Enables or disables the use of image verbalization. The default is <code>False</code> , which <i>enables</i> image verbalization. Set to <code>True</code> to <i>disable</i> image verbalization.	Boolean	No	No
<code>chat_completion_model</code>	A chat completion model that verbalizes images or extracts content. Supported models are <code>gpt-4o</code> , <code>gpt-4o-mini</code> , <code>gpt-4.1</code> , <code>gpt-4.1-mini</code> , <code>gpt-4.1-nano</code> , <code>gpt-5</code> , <code>gpt-5-mini</code> , and <code>gpt-5-nano</code> . The GenAI Prompt skill will be included in the generated skillset. Setting this parameter also requires that <code>disable_image_verbalization</code> is set to <code>False</code> .	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No
<code>embedding_model</code>	A text embedding model that vectorizes text and image content during indexing and at query time. Supported models are <code>text-embedding-ada-002</code> , <code>text-embedding-3-small</code> , and <code>text-embedding-3-large</code> . The Azure OpenAI Embedding skill will be included in the generated skillset, and the Azure OpenAI vectorizer will be included in the generated index.	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No
<code>content_extraction_mode</code>	Controls how content is extracted from files. The default is <code>minimal</code> , which uses standard content extraction for text and images. Set to <code>standard</code> for advanced document cracking and chunking using the Azure	String	No	No

Name	Description	Type	Editable	Required
	<p>Content Understanding skill, which will be included in the generated skillset. For standard only, the ai_services and asset_store parameters are specifiable.</p>			
ai_services	A Microsoft Foundry resource to access Azure Content Understanding in Foundry Tools. Setting this parameter requires that content_extraction_mode is set to standard.	Object	Only api_key is editable	Yes
asset_store	A blob container to store extracted images. Setting this parameter requires that content_extraction_mode is set to standard.	Object	No	No
ingestion_schedule	Adds scheduling information to the generated indexer. You can also add a schedule later to automate data refresh.	Object	Yes	No
ingestion_permission_options	The document-level permissions to ingest from select knowledge sources: either ADLS Gen2 or indexed SharePoint. If you specify user_ids, group_ids, or rbac_scope, the generated ADLS Gen2 indexer or SharePoint indexer will include the ingested permissions.	Array	No	No

Check ingestion status

Run the following code to monitor ingestion progress and health, including indexer status for knowledge sources that generate an indexer pipeline and populate a search index.

Python

```
# Check knowledge source ingestion status
import requests
```

```

import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}/status"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))

```

A response for a request that includes ingestion parameters and is actively ingesting content might look like the following example.

JSON

```
{
  "synchronizationStatus": "active", // creating, active, deleting
  "synchronizationInterval" : "1d", // null if no schedule
  "currentSynchronizationState" : { // spans multiple indexer "runs"
    "startTime": "2025-10-27T19:30:00Z",
    "itemUpdatesProcessed": 1100,
    "itemsUpdatesFailed": 100,
    "itemsSkipped": 1100,
  },
  "lastSynchronizationState" : { // null on first sync
    "startTime": "2025-10-27T19:30:00Z",
    "endTime": "2025-10-27T19:40:01Z", // this value appears on the activity record
on each /retrieve
    "itemUpdatesProcessed": 1100,
    "itemsUpdatesFailed": 100,
    "itemsSkipped": 1100,
  },
  "statistics": { // null on first sync
    "totalSynchronization": 25,
    "averageSynchronizationDuration": "00:15:20",
    "averageItemsProcessedPerSynchronization" : 500
  }
}
```

Review the created objects

When you create a blob knowledge source, your search service also creates an indexer, index, skillset, and data source. We don't recommend that you edit these objects, as introducing an error or incompatibility can break the pipeline.

After you create a knowledge source, the response lists the created objects. These objects are created according to a fixed template, and their names are based on the name of the knowledge source. You can't change the object names.

We recommend using the Azure portal to validate output creation. The workflow is:

1. Check the indexer for success or failure messages. Connection or quota errors appear here.
2. Check the index for searchable content. Use Search Explorer to run queries.
3. Check the skillset to learn how your content is chunked and optionally vectorized.
4. Check the data source for connection details. Our example uses API keys for simplicity, but you can use Microsoft Entra ID for authentication and role-based access control for authorization.

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
    "@odata.context": "https://my-search-  
service.search.windows.net/$metadata#knowledgebases(name)",  
    "value": [  
        {  
            "name": "my-kb"  
        },  
        {  
            "name": "my-kb-2"  
        }  
    ]  
}
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition  
import requests  
import json  
  
endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"  
params = {"api-version": "2025-11-01-preview"}  
headers = {"api-key": "{api_key}"}  
  
response = requests.get(endpoint, params = params, headers = headers)  
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
    "name": "my-kb",  
    "description": null,  
    "retrievalInstructions": null,  
    "answerInstructions": null,  
    "outputMode": null,  
    "knowledgeSources": [  
        {  
            "name": "my-blob-ks",  
        }  
    ],  
    "models": [],  
    "encryptionKey": null,  
    "retrievalReasoningEffort": {  
        "kind": "low"  
    }  
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_base("knowledge_base_name")
print(f"Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_source("knowledge_source_name")
print(f"Knowledge source deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Azure AI Search blob knowledge source Python sample ↗](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\) ↗](#)
- [Azure OpenAI demo featuring agentic retrieval ↗](#)

Last updated on 11/21/2025

Create a OneLake knowledge source

(!) Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Use a *OneLake knowledge source* to index and query Microsoft OneLake files in an agentic retrieval pipeline. [Knowledge sources](#) are created independently, referenced in a [knowledge base](#), and used as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

When you create a OneLake knowledge source, you specify an external data source, models, and properties to automatically generate the following Azure AI Search objects:

- A data source that represents a lakehouse.
- A skillset that chunks and optionally vectorizes multimodal content from the lakehouse.
- An index that stores enriched content and meets the criteria for agentic retrieval.
- An indexer that uses the previous objects to drive the indexing and enrichment pipeline.

The generated indexer conforms to the *OneLake indexer*, whose prerequisites, supported tasks, supported document formats, supported shortcuts, and limitations also apply to OneLake knowledge sources. For more information, see the [OneLake indexer documentation](#).

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#).
- Completion of the [OneLake indexer prerequisites](#).
- Completion of the [OneLake indexer data preparation](#).
- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a OneLake knowledge source.

JSON

```
{
  "name": "my-onelake-ks",
  "kind": "indexedOneLake",
  "description": "A sample indexed OneLake knowledge source.",
  "encryptionKey": null,
  "indexedOneLakeParameters": {
    "fabricWorkspaceId": "<REDACTED>",
    "lakehouseId": "<REDACTED>",
    "targetPath": null,
    "ingestionParameters": {
      "disableImageVerbalization": false,
      "ingestionPermissionOptions": [],
      "contentExtractionMode": "standard",
      "identity": null,
```

```
"embeddingModel": {  
    "kind": "azureOpenAI",  
    "azureOpenAIParameters": {  
        "resourceUri": "<REDACTED>",  
        "deploymentId": "text-embedding-3-large",  
        "apiKey": "<REDACTED>",  
        "modelName": "text-embedding-3-large"  
    }  
},  
"chatCompletionModel": {  
    "kind": "azureOpenAI",  
    "azureOpenAIParameters": {  
        "resourceUri": "<REDACTED>",  
        "deploymentId": "gpt-5-mini",  
        "apiKey": "<REDACTED>",  
        "modelName": "gpt-5-mini"  
    }  
},  
"ingestionSchedule": null,  
"aiServices": {  
    "uri": "<REDACTED>",  
    "apiKey": "<REDACTED>"  
},  
"createdResources": {  
    "datasource": "my-onelake-ks-datasource",  
    "indexer": "my-onelake-ks-indexer",  
    "skillset": "my-onelake-ks-skillset",  
    "index": "my-onelake-ks-index"  
}  
}  
}  
}
```

ⓘ Note

Sensitive information is redacted. The generated resources appear at the end of the response.

Create a knowledge source

Run the following code to create a OneLake knowledge source.

Python

```
# Create a OneLake knowledge source  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient  
from azure.search.documents.indexes.models import IndexedOneLakeKnowledgeSource,  
IndexedOneLakeKnowledgeSourceParameters, KnowledgeBaseAzureOpenAIModel,
```

```

AzureOpenAIVectorizerParameters, KnowledgeSourceAzureOpenAIVectorizer,
KnowledgeSourceContentExtractionMode, KnowledgeSourceIngestionParameters

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))

knowledge_source = IndexedOneLakeKnowledgeSource(
    name = "my-onelake-ks",
    description= "This knowledge source pulls content from a lakehouse.",
    encryption_key = None,
    indexed_one_lake_parameters = IndexedOneLakeKnowledgeSourceParameters(
        fabric_workspace_id = "fabric_workspace_id",
        lakehouse_id = "lakehouse_id",
        target_path = None,
        ingestion_parameters = KnowledgeSourceIngestionParameters(
            identity = None,
            disable_image_verbalization = False,
            chat_completion_model = KnowledgeBaseAzureOpenAIModel(
                azure_open_ai_parameters = AzureOpenAIVectorizerParameters(
                    # TRIMMED FOR BREVITY
                )
            ),
            embedding_model = KnowledgeSourceAzureOpenAIVectorizer(
                azure_open_ai_parameters=AzureOpenAIVectorizerParameters(
                    # TRIMMED FOR BREVITY
                )
            ),
            content_extraction_mode = KnowledgeSourceContentExtractionMode.MINIMAL,
            ingestion_schedule = None,
            ingestion_permission_options = None
        )
    )
)
)

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated
successfully.")

```

Source-specific properties

You can pass the following properties to create a OneLake knowledge source.

[] Expand table

Name	Description	Type	Editable	Required
name	The name of the knowledge source, which must be unique within the knowledge sources collection and follow the naming guidelines for objects in Azure AI Search.	String	Yes	Yes

Name	Description	Type	Editable	Required
<code>description</code>	A description of the knowledge source.	String	Yes	No
<code>encryption</code>	A customer-managed key to encrypt sensitive information in both the knowledge source and the generated objects.	Object	Yes	No
<code>indexed_one_lake_parameters</code>	Parameters specific to OneLake knowledge sources: <code>fabric_workspace_id</code> , <code>lakehouse_id</code> , and <code>target_path</code> .	Object		Yes
<code>fabric_workspace_id</code>	The GUID of the workspace that contains the lakehouse.	String	No	Yes
<code>lakehouse_id</code>	The GUID of the lakehouse.	String	No	Yes
<code>target_path</code>	A folder or shortcut within the lakehouse. When unspecified, the entire lakehouse is indexed.	String	No	No

ingestionParameters properties

For indexed knowledge sources only, you can pass the following `ingestionParameters` properties to control how content is ingested and processed.

[\[+\]](#) Expand table

Name	Description	Type	Editable	Required
<code>identity</code>	A managed identity to use in the generated indexer.	Object	Yes	No
<code>disable_image_verbalization</code>	Enables or disables the use of image verbalization. The default is <code>False</code> , which enables image verbalization. Set to <code>True</code> to disable image verbalization.	Boolean	No	No
<code>chat_completion_model</code>	A chat completion model that verbalizes images or extracts content. Supported models are <code>gpt-40</code> , <code>gpt-40-mini</code> , <code>gpt-4.1</code> , <code>gpt-4.1-mini</code> , <code>gpt-4.1-nano</code> , <code>gpt-5</code> , <code>gpt-5-mini</code> , and	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No

Name	Description	Type	Editable	Required
	<p><code>gpt-5-nano</code>. The GenAI Prompt skill will be included in the generated skillset. Setting this parameter also requires that <code>disable_image_verbalization</code> is set to <code>False</code>.</p>			
<code>embedding_model</code>	<p>A text embedding model that vectorizes text and image content during indexing and at query time. Supported models are <code>text-embedding-ada-002</code>, <code>text-embedding-3-small</code>, and <code>text-embedding-3-large</code>. The Azure OpenAI Embedding skill will be included in the generated skillset, and the Azure OpenAI vectorizer will be included in the generated index.</p>	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No
<code>content_extraction_mode</code>	<p>Controls how content is extracted from files. The default is <code>minimal</code>, which uses standard content extraction for text and images. Set to <code>standard</code> for advanced document cracking and chunking using the Azure Content Understanding skill, which will be included in the generated skillset. For <code>standard</code> only, the <code>ai_services</code> and <code>asset_store</code> parameters are specifiable.</p>	String	No	No
<code>ai_services</code>	<p>A Microsoft Foundry resource to access Azure Content Understanding in Foundry Tools. Setting this parameter requires that <code>content_extraction_mode</code> is set to <code>standard</code>.</p>	Object	Only <code>api_key</code> is editable	Yes
<code>asset_store</code>	<p>A blob container to store extracted images. Setting this parameter requires that</p>	Object	No	No

Name	Description	Type	Editable	Required
	<code>content_extraction_mode</code> is set to <code>standard</code> .			
<code>ingestion_schedule</code>	Adds scheduling information to the generated indexer. You can also add a schedule later to automate data refresh.	Object	Yes	No
<code>ingestion_permission_options</code>	The document-level permissions to ingest from select knowledge sources: either ADLS Gen2 or indexed SharePoint . If you specify <code>user_ids</code> , <code>group_ids</code> , or <code>rbac_scope</code> , the generated ADLS Gen2 indexer or SharePoint indexer will include the ingested permissions.	Array	No	No

Check ingestion status

Run the following code to monitor ingestion progress and health, including indexer status for knowledge sources that generate an indexer pipeline and populate a search index.

Python

```
# Check knowledge source ingestion status
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}/status"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

A response for a request that includes ingestion parameters and is actively ingesting content might look like the following example.

JSON

```
{
  "synchronizationStatus": "active", // creating, active, deleting
  "synchronizationInterval" : "1d", // null if no schedule
  "currentSynchronizationState" : { // spans multiple indexer "runs"
```

```
"startTime": "2025-10-27T19:30:00Z",
"itemUpdatesProcessed": 1100,
"itemsUpdatesFailed": 100,
"itemsSkipped": 1100,
},
"lastSynchronizationState" : { // null on first sync
  "startTime": "2025-10-27T19:30:00Z",
  "endTime": "2025-10-27T19:40:01Z", // this value appears on the activity record
on each /retrieve
  "itemUpdatesProcessed": 1100,
  "itemsUpdatesFailed": 100,
  "itemsSkipped": 1100,
},
"statistics": { // null on first sync
  "totalSynchronization": 25,
  "averageSynchronizationDuration": "00:15:20",
  "averageItemsProcessedPerSynchronization" : 500
}
}
```

Review the created objects

When you create a OneLake knowledge source, your search service also creates an indexer, index, skillset, and data source. We don't recommend that you edit these objects, as introducing an error or incompatibility can break the pipeline.

After you create a knowledge source, the response lists the created objects. These objects are created according to a fixed template, and their names are based on the name of the knowledge source. You can't change the object names.

We recommend using the Azure portal to validate output creation. The workflow is:

1. Check the indexer for success or failure messages. Connection or quota errors appear here.
2. Check the index for searchable content. Use Search Explorer to run queries.
3. Check the skillset to learn how your content is chunked and optionally vectorized.
4. Check the data source for connection details. Our example uses API keys for simplicity, but you can use Microsoft Entra ID for authentication and role-based access control for authorization.

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

For any knowledge base that specifies a OneLake knowledge source, be sure to set `includeReferenceSourceData` to `true`. This step is necessary for pulling the source document URL into the citation.

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
    "@odata.context": "https://my-search-
service.search.windows.net/$metadata#knowledgebases(name)",
    "value": [
        {
            "name": "my-kb"
        },
        {
            "name": "my-kb-2"
        }
    ]
}
```

```
    ]  
}
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition  
import requests  
import json  
  
endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"  
params = {"api-version": "2025-11-01-preview"}  
headers = {"api-key": "{api_key}"}  
  
response = requests.get(endpoint, params = params, headers = headers)  
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
  "name": "my-kb",  
  "description": null,  
  "retrievalInstructions": null,  
  "answerInstructions": null,  
  "outputMode": null,  
  "knowledgeSources": [  
    {  
      "name": "my-blob-ks",  
    }  
  ],  
  "models": [],  
  "encryptionKey": null,  
  "retrievalReasoningEffort": {  
    "kind": "low"  
  }  
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient  
  
index_client = SearchIndexClient(endpoint = "search_url", credential =  
AzureKeyCredential("api_key"))
```

```
index_client.delete_knowledge_base("knowledge_base_name")
print(f"Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_source("knowledge_source_name")
print(f"Knowledge source deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Last updated on 11/21/2025

Create an indexed SharePoint knowledge source

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Use an *indexed SharePoint knowledge source* to index and query SharePoint content in an agentic retrieval pipeline. [Knowledge sources](#) are created independently, referenced in a [knowledge base](#), and used as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

When you create an indexed SharePoint knowledge source, you specify a SharePoint connection string, models, and properties to automatically generate the following Azure AI Search objects:

- A data source that points to SharePoint sites.
- A skillset that chunks and optionally vectorizes multimodal content.
- An index that stores enriched content and meets the criteria for agentic retrieval.
- An indexer that uses the previous objects to drive the indexing and enrichment pipeline.

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#).
- Completion of the [SharePoint indexer prerequisites](#).
- Completion of three SharePoint indexer configuration steps:
 - [Step 1: Enable a managed identity for Azure AI Search](#)
 - [Step 2: Choose between delegated or application permissions](#)
 - [Step 3: Application registration step for Microsoft Entra ID authentication](#)
- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for an indexed SharePoint knowledge source.

JSON

```
{
  "name": "my-indexed-sharepoint-ks",
  "kind": "indexedSharePoint",
  "description": "A sample indexed SharePoint knowledge source",
  "encryptionKey": null,
  "indexedSharePointParameters": {
    "connectionString": "<redacted>",
    "containerName": "defaultSiteLibrary",
    "query": null,
    "ingestionParameters": {
      "disableImageVerbalization": false,
```

```
"ingestionPermissionOptions": [],
"contentExtractionMode": "minimal",
"identity": null,
"embeddingModel": {
    "kind": "azureOpenAI",
    "azureOpenAIParameters": {
        "resourceUri": "<redacted>",
        "deploymentId": "text-embedding-3-large",
        "apiKey": "<redacted>",
        "modelName": "text-embedding-3-large",
        "authIdentity": null
    }
},
"chatCompletionModel": null,
"ingestionSchedule": null,
"assetStore": null,
"aiServices": null
},
"createdResources": {
    "datasource": "my-indexed-sharepoint-ks-datasource",
    "indexer": "my-indexed-sharepoint-ks-indexer",
    "skillset": "my-indexed-sharepoint-ks-skillset",
    "index": "my-indexed-sharepoint-ks-index"
}
},
"indexedOneLakeParameters": null
}
```

ⓘ Note

Sensitive information is redacted. The generated resources appear at the end of the response.

Create a knowledge source

Run the following code to create an indexed SharePoint knowledge source.

Python

```
# Create an indexed SharePoint knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import IndexedSharePointKnowledgeSource,
IndexedSharePointKnowledgeSourceParameters, KnowledgeBaseAzureOpenAIModel,
AzureOpenAIVectorizerParameters, KnowledgeSourceAzureOpenAIVectorizer,
KnowledgeSourceContentExtractionMode, KnowledgeSourceIngestionParameters

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
```

```

knowledge_source = IndexedSharePointKnowledgeSource(
    name = "my-indexed-sharepoint-ks",
    description = "A sample indexed SharePoint knowledge source.",
    encryption_key = None,
    indexed_share_point_parameters = IndexedSharePointKnowledgeSourceParameters(
        connection_string = "connection_string",
        container_name = "defaultSiteLibrary",
        query = None,
        ingestion_parameters = KnowledgeSourceIngestionParameters(
            identity = None,
            disable_image_verbalization = False,
            chat_completion_model = KnowledgeBaseAzureOpenAIModel(
                azure_open_ai_parameters = AzureOpenAIVectorizerParameters(
                    # TRIMMED FOR BREVITY
                )
            ),
            embedding_model = KnowledgeSourceAzureOpenAIVectorizer(
                azure_open_ai_parameters=AzureOpenAIVectorizerParameters(
                    # TRIMMED FOR BREVITY
                )
            ),
            content_extraction_mode = KnowledgeSourceContentExtractionMode.MINIMAL,
            ingestion_schedule = None,
            ingestion_permission_options = None
        )
    )
)
)

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated successfully.")

```

Source-specific properties

You can pass the following properties to create an indexed SharePoint knowledge source.

[] Expand table

Name	Description	Type	Editable	Required
name	The name of the knowledge source, which must be unique within the knowledge sources collection and follow the naming guidelines for objects in Azure AI Search.	String	No	Yes
description	A description of the knowledge source.	String	Yes	No

Name	Description	Type	Editable	Required
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in both the knowledge source and the generated objects.	Object	Yes	No
<code>indexed_share_point_parameters</code>	Parameters specific to indexed SharePoint knowledge sources: <code>connection_string</code> , <code>container_name</code> , and <code>query</code> .	Object	No	No
<code>connection_string</code>	The connection string to a SharePoint site. For more information, see Connection string syntax .	String	Yes	Yes
<code>container_name</code>	The SharePoint library to access. Use <code>defaultSiteLibrary</code> to index content from the site's default document library or <code>allSiteLibraries</code> to index content from every document library in the site. Ignore <code>useQuery</code> for now.	String	No	Yes
<code>query</code>	Ignore for now.	String	Yes	No

ingestion_parameters properties

For indexed knowledge sources only, you can pass the following `ingestionParameters` properties to control how content is ingested and processed.

[Expand table](#)

Name	Description	Type	Editable	Required
<code>identity</code>	A managed identity to use in the generated indexer.	Object	Yes	No
<code>disable_image_verbalization</code>	Enables or disables the use of image verbalization. The default is <code>False</code> , which <i>enables</i> image verbalization. Set to <code>True</code> to <i>disable</i> image verbalization.	Boolean	No	No
<code>chat_completion_model</code>	A chat completion model that verbalizes images or extracts content. Supported models are <code>gpt-40</code> , <code>gpt-40-mini</code> , <code>gpt-</code>	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No

Name	Description	Type	Editable	Required
	<p>4.1, gpt-4.1-mini, gpt-4.1-nano, gpt-5, gpt-5-mini, and gpt-5-nano. The GenAI Prompt skill will be included in the generated skillset. Setting this parameter also requires that <code>disable_image_verbalization</code> is set to <code>False</code>.</p>			
<code>embedding_model</code>	<p>A text embedding model that vectorizes text and image content during indexing and at query time. Supported models are <code>text-embedding-ada-002</code>, <code>text-embedding-3-small</code>, and <code>text-embedding-3-large</code>. The Azure OpenAI Embedding skill will be included in the generated skillset, and the Azure OpenAI vectorizer will be included in the generated index.</p>	Object	Only <code>api_key</code> and <code>deployment_name</code> are editable	No
<code>content_extraction_mode</code>	<p>Controls how content is extracted from files. The default is <code>minimal</code>, which uses standard content extraction for text and images. Set to <code>standard</code> for advanced document cracking and chunking using the Azure Content Understanding skill, which will be included in the generated skillset. For <code>standard</code> only, the <code>ai_services</code> and <code>asset_store</code> parameters are specifiable.</p>	String	No	No
<code>ai_services</code>	<p>A Microsoft Foundry resource to access Azure Content Understanding in Foundry Tools. Setting this parameter requires that <code>content_extraction_mode</code> is set to <code>standard</code>.</p>	Object	Only <code>api_key</code> is editable	Yes

Name	Description	Type	Editable	Required
<code>asset_store</code>	A blob container to store extracted images. Setting this parameter requires that <code>content_extraction_mode</code> is set to <code>standard</code> .	Object	No	No
<code>ingestion_schedule</code>	Adds scheduling information to the generated indexer. You can also add a schedule later to automate data refresh.	Object	Yes	No
<code>ingestion_permission_options</code>	The document-level permissions to ingest from select knowledge sources: either ADLS Gen2 or indexed SharePoint . If you specify <code>user_ids</code> , <code>group_ids</code> , or <code>rbac_scope</code> , the generated ADLS Gen2 indexer or SharePoint indexer will include the ingested permissions.	Array	No	No

Check ingestion status

Run the following code to monitor ingestion progress and health, including indexer status for knowledge sources that generate an indexer pipeline and populate a search index.

Python

```
# Check knowledge source ingestion status
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}/status"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

A response for a request that includes ingestion parameters and is actively ingesting content might look like the following example.

JSON

```
{
  "synchronizationStatus": "active", // creating, active, deleting
  "synchronizationInterval" : "1d", // null if no schedule
  "currentSynchronizationState" : { // spans multiple indexer "runs"
    "startTime": "2025-10-27T19:30:00Z",
    "itemUpdatesProcessed": 1100,
    "itemsUpdatesFailed": 100,
    "itemsSkipped": 1100,
  },
  "lastSynchronizationState" : { // null on first sync
    "startTime": "2025-10-27T19:30:00Z",
    "endTime": "2025-10-27T19:40:01Z", // this value appears on the activity record
    on each /retrieve
    "itemUpdatesProcessed": 1100,
    "itemsUpdatesFailed": 100,
    "itemsSkipped": 1100,
  },
  "statistics": { // null on first sync
    "totalSynchronization": 25,
    "averageSynchronizationDuration": "00:15:20",
    "averageItemsProcessedPerSynchronization" : 500
  }
}
```

Review the created objects

When you create an indexed SharePoint knowledge source, your search service also creates an indexer, index, skillset, and data source. We don't recommend that you edit these objects, as introducing an error or incompatibility can break the pipeline.

After you create a knowledge source, the response lists the created objects. These objects are created according to a fixed template, and their names are based on the name of the knowledge source. You can't change the object names.

We recommend using the Azure portal to validate output creation. The workflow is:

1. Check the indexer for success or failure messages. Connection or quota errors appear here.
2. Check the index for searchable content. Use Search Explorer to run queries.
3. Check the skillset to learn how your content is chunked and optionally vectorized.
4. Check the data source for connection details. Our example uses API keys for simplicity, but you can use Microsoft Entra ID for authentication and role-based access control for authorization.

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

For any knowledge base that specifies an indexed SharePoint knowledge source, be sure to set `includeReferenceSourceData` to `true`. This step is necessary for pulling the source document URL into the citation.

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
    "@odata.context": "https://my-search-
service.search.windows.net/$metadata#knowledgebases(name)",
    "value": [
        {
            "name": "my-kb"
        },
    ],
}
```

```
{  
    "name": "my-kb-2"  
}  
]  
}
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition  
import requests  
import json  
  
endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"  
params = {"api-version": "2025-11-01-preview"}  
headers = {"api-key": "{api_key}"}  
  
response = requests.get(endpoint, params = params, headers = headers)  
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
    "name": "my-kb",  
    "description": null,  
    "retrievalInstructions": null,  
    "answerInstructions": null,  
    "outputMode": null,  
    "knowledgeSources": [  
        {  
            "name": "my-blob-ks",  
        }  
    ],  
    "models": [],  
    "encryptionKey": null,  
    "retrievalReasoningEffort": {  
        "kind": "low"  
    }  
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient
```

```
index_client = SearchIndexClient(endpoint = "search_url", credential =  
    AzureKeyCredential("api_key"))  
index_client.delete_knowledge_base("knowledge_base_name")  
print(f"Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient  
  
index_client = SearchIndexClient(endpoint = "search_url", credential =  
    AzureKeyCredential("api_key"))  
index_client.delete_knowledge_source("knowledge_source_name")  
print(f"Knowledge source deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Last updated on 11/21/2025

Create a remote SharePoint knowledge source

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

A *remote SharePoint knowledge source* uses the [Copilot Retrieval API](#) to query textual content directly from SharePoint in Microsoft 365, returning results to the agentic retrieval engine for merging, ranking, and response formulation. There's no search index used by this knowledge source, and only textual content is queried.

At query time, the remote SharePoint knowledge source calls the Copilot Retrieval API on behalf of the user identity, so no connection strings are needed in the knowledge source definition. All content to which a user has access is in-scope for knowledge retrieval. To limit sites or constrain search, set a [filter expression](#). Your Azure tenant and the Microsoft 365 tenant must use the same Microsoft Entra ID tenant, and the caller's identity must be recognized by both tenants.

- You can use filters to scope search by URLs, date ranges, file types, and other metadata.
- SharePoint permissions and Purview labels are honored in requests for content.
- Usage is billed through Microsoft 365 and a Copilot license.

Like any other knowledge source, you specify a remote SharePoint knowledge source in a [knowledge base](#) and use the results as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#).
- SharePoint in a Microsoft 365 tenant that's under the same Microsoft Entra ID tenant as Azure.
- A personal access token for local development or a user's identity from a client application.

- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible.

For local development, the agentic retrieval engine uses your access token to call SharePoint on your behalf. For more information about using a personal access token on requests, see [Connect to Azure AI Search](#).

Limitations

The following limitations in the [Copilot Retrieval API](#) apply to remote SharePoint knowledge sources.

- There's no support for Copilot connectors or OneDrive content. Content is retrieved from SharePoint sites only.
- Limit of 200 requests per user per hour.
- Query character limit of 1,500 characters.
- Hybrid queries are only supported for the following file extensions: .doc, .docx, .pptx, .pdf, .aspx, and .one.
- Multimodal retrieval (nontextual content, including tables, images, and charts) isn't supported.
- Maximum of 25 results from a query.
- Results are returned by Copilot Retrieval API as unordered.
- Invalid Keyword Query Language (KQL) filter expressions are ignored and the query continues to execute without the filter.

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
```

```
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a remote SharePoint knowledge source.

JSON

```
{
  "name": "my-sharepoint-ks",
  "kind": "remoteSharePoint",
  "description": "A sample remote SharePoint knowledge source",
  "encryptionKey": null,
  "remoteSharePointParameters": {
    "filterExpression": "filetype:docx",
    "containerTypeId": null,
    "resourceMetadata": [
      "Author",
      "Title"
    ]
  }
}
```

Create a knowledge source

Run the following code to create a remote SharePoint knowledge source.

API keys are used for your client connection to Azure AI Search and Azure OpenAI. Your access token is used by Azure AI Search to connect to SharePoint in Microsoft 365 on your behalf. You

can only retrieve content that you're permitted to access. For more information about getting a personal access token and other values, see [Connect to Azure AI Search](#).

! Note

You can also use your personal access token to access Azure AI Search and Azure OpenAI if you [set up role assignments on each resource](#). Using API keys allows you to omit this step in this example.

Python

```
# Create a remote SharePoint knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import RemoteSharePointKnowledgeSource,
RemoteSharePointKnowledgeSourceParameters

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))

knowledge_source = RemoteSharePointKnowledgeSource(
    name = "my-remote-sharepoint-ks",
    description= "This knowledge source queries .docx files in a trusted Microsoft
365 tenant.",
    encryption_key = None,
    remote_share_point_parameters = RemoteSharePointKnowledgeSourceParameters(
        filter_expression = "filetype:docx",
        resource_metadata = ["Author", "Title"],
        container_type_id = None
    )
)

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated
successfully.")
```

Source-specific properties

You can pass the following properties to create a remote SharePoint knowledge source.

[+] Expand table

Name	Description	Type	Editable	Required
name	The name of the knowledge source, which must be unique within the knowledge sources collection and	String	No	Yes

Name	Description	Type	Editable	Required
	follow the naming guidelines for objects in Azure AI Search.			
<code>description</code>	A description of the knowledge source.	String	Yes	No
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in the knowledge source.	Object	Yes	No
<code>remote_share_point_parameters</code>	Parameters specific to remote SharePoint knowledge sources: <code>filter_expression</code> , <code>resource_metadata</code> , and <code>container_type_id</code> .	Object	No	No
<code>filter_expression</code>	An expression written in the SharePoint KQL , which is used to specify sites and paths to content.	String	Yes	No
<code>resource_metadata</code>	A comma-delimited list of standard metadata fields: author, file name, creation date, content type, and file type.	Array	Yes	No
<code>container_type_id</code>	Container ID for the SharePoint Embedded connection. When unspecified, SharePoint Online is used.	String	Yes	No

Filter expression examples

Not all SharePoint properties are supported in the `filterExpression`. For a list of supported properties, see the [API reference](#). Here's some more information about queryable properties that you can use in filter: [queryable properties](#).

Learn more about [KQL filters](#) in the syntax reference.

[Expand table](#)

Example	Filter expression
Filter to a single site by ID	<code>"filterExpression": "SiteID:\\"00aa00aa-bb11-cc22-dd33-44ee44ee44ee\\""</code>
Filter to multiple sites by ID	<code>"filterExpression": "SiteID:\\"00aa00aa-bb11-cc22-dd33-44ee44ee44ee\\" OR SiteID:\\"11bb11bb-cc22-dd33-ee44-55ff55ff55ff\\\""</code>
Filter to files under a specific path	<code>"filterExpression": "Path:\\"https://my-demosharepoint.sharepoint.com/sites/mysite/Shared Documents/en/mydocs\\""</code>

Example	Filter expression
Filter to a specific date range	<code>"filterExpression": "LastModifiedTime >= 2024-07-22 AND LastModifiedTime <= 2025-01-08"</code>
Filter to files of a specific file type	<code>"filterExpression": "FileExtension:\"docx\" OR FileExtension:\"pdf\" OR FileExtension:\"pptx\""</code>
Filter to files of a specific information protection label	<code>"filterExpression": "InformationProtectionLabelId:\"f0ddcc93-d3c0-4993-b5cc-76b0a283e252\""</code>

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Query a knowledge base

The [retrieve action](#) on the knowledge base provides the user identity that authorizes access to content in Microsoft 365.

Azure AI Search uses the access token to call the Copilot Retrieval API on behalf of the user identity. The access token is provided in the retrieve endpoint as a `x-ms-query-source-authorization` HTTP header.

Python

```
from azure.identity import DefaultAzureCredential, get_bearer_token_provider
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.knowledgebases import KnowledgeBaseRetrievalClient
from azure.search.documents.knowledgebases.models import KnowledgeBaseMessage,
KnowledgeBaseMessageTextContent, KnowledgeBaseRetrievalRequest,
RemoteSharePointKnowledgeSourceParams

# Get access token
identity_token_provider = get_bearer_token_provider(DefaultAzureCredential(),
"https://search.azure.com/.default")
token = identity_token_provider()

# Create knowledge base retrieval client
kb_client = KnowledgeBaseRetrievalClient(endpoint = "search_url",
knowledge_base_name = "knowledge_base_name", credential =
AzureKeyCredential("api_key"))
```

```

# Create retrieval request
request = KnowledgeBaseRetrievalRequest(
    include_activity = True,
    messages = [
        KnowledgeBaseMessage(role = "user", content =
[KnowledgeBaseMessageTextContent(text = "What was covered in the keynote doc for
Ignite 2024?")])
    ],
    knowledge_source_params = [
        RemoteSharePointKnowledgeSourceParams(
            knowledge_source_name = "my-remote-sharepoint-ks",
            filter_expression_add_on = "ModifiedBy:\\"Adele Vance\\\"",
            include_references = True,
            include_reference_source_data = True
        )
    ]
)

# Pass access token to retrieve from knowledge base
result = kb_client.retrieve(retrieval_request = request,
x_ms_query_source_authorization = token)
print(result.response[0].content[0].text)

```

The retrieve request also takes a [KQL filter](#) (`filter_expression_add_on`) if you want to apply constraints at query time. If you specify `filter_expression_add_on` on both the knowledge source and knowledge base retrieve action, the filters are AND'd together.

Queries asking questions about the content itself are more effective than questions about where a file is located or when it was last updated. For example, if you ask, "Where is the keynote doc for Ignite 2024", you might get "No relevant content was found for your query" because the content itself doesn't disclose its location. A filter on metadata is a better solution for file location or date-specific queries.

A better question to ask is, "What is the keynote doc for Ignite 2024". The response includes the synthesized answer, query activity and token counts, plus the URL and other metadata.

JSON

```
{
    "resourceMetadata": {
        "Author": "Nutan Amarathunga; Nurul Izzati",
        "Title": "Ignite 2024 Keynote Address"
    },
    "rerankerScore": 2.489522,
    "webUrl": "https://contoso-
my.sharepoint.com/keynotes/nuamarth_contoso_com/Documents/Keynote-Ignite-2024.docx",
    "searchSensitivityLabelInfo": {
        "displayName": "Confidential\\Contoso Extended",
        "sensitivityLabelId": "aaaaaaaa-0b0b-1c1c-2d2d-333333333333",
        "tooltip": "Data is classified and protected. Contoso Full Time Employees"
    }
}
```

(FTE) and non-employees can edit, reply, forward and print. Recipient can unprotect content with the right justification.",

```
    "priority": 5,  
    "color": "#FF8C00",  
    "isEncrypted": true  
}
```

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases  
import requests  
import json  
  
endpoint = "{search_url}/knowledgebases"  
params = {"api-version": "2025-11-01-preview", "$select": "name"}  
headers = {"api-key": "{api_key}"}  
  
response = requests.get(endpoint, params = params, headers = headers)  
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
    "@odata.context": "https://my-search-  
service.search.windows.net/$metadata#knowledgebases(name)",  
    "value": [  
        {  
            "name": "my-kb"  
        },  
        {  
            "name": "my-kb-2"  
        }  
    ]}
```

```
    ]  
}
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition  
import requests  
import json  
  
endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"  
params = {"api-version": "2025-11-01-preview"}  
headers = {"api-key": "{api_key}"}  
  
response = requests.get(endpoint, params = params, headers = headers)  
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{  
  "name": "my-kb",  
  "description": null,  
  "retrievalInstructions": null,  
  "answerInstructions": null,  
  "outputMode": null,  
  "knowledgeSources": [  
    {  
      "name": "my-blob-ks",  
    }  
  ],  
  "models": [],  
  "encryptionKey": null,  
  "retrievalReasoningEffort": {  
    "kind": "low"  
  }  
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient  
  
index_client = SearchIndexClient(endpoint = "search_url", credential =  
AzureKeyCredential("api_key"))
```

```
index_client.delete_knowledge_base("knowledge_base_name")
print(f"Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_source("knowledge_source_name")
print(f"Knowledge source deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Last updated on 11/21/2025

Create a search index knowledge source

(!) Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

A *search index knowledge source* specifies a connection to an Azure AI Search index that provides searchable content in an agentic retrieval pipeline. [Knowledge sources](#) are created independently, referenced in a [knowledge base](#), and used as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#).
- A search index containing plain text or vector content with a semantic configuration.
[Review the index criteria for agentic retrieval](#). The index must be on the same search service as the knowledge base.
- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).

(!) Note

Although you can use the Azure portal to create search index knowledge sources, the portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a search index knowledge source. Notice that the knowledge source specifies a single index name and which fields in the index to include in the query.

JSON

```
{
  "name": "my-search-index-ks",
  "kind": "searchIndex",
  "description": "A sample search index knowledge source.",
  "encryptionKey": null,
  "searchIndexParameters": {
    "searchIndexName": "my-search-index",
    "semanticConfigurationName": null,
    "sourceDataFields": [],
    "searchFields": []
  }
}
```

Create a knowledge source

Run the following code to create a search index knowledge source.

Python

```
# Create a search index knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import SearchIndexKnowledgeSource,
SearchIndexKnowledgeSourceParameters, SearchIndexFieldReference

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))

knowledge_source = SearchIndexKnowledgeSource(
    name = "my-search-index-ks",
    description= "This knowledge source pulls from an existing index designed for
    agentic retrieval.",
    encryption_key = None,
    search_index_parameters = SearchIndexKnowledgeSourceParameters(
        search_index_name = "search_index_name",
        semantic_configuration_name = "semantic_configuration_name",
        source_data_fields = [
            SearchIndexFieldReference(name="description"),
            SearchIndexFieldReference(name="category"),
        ],
        search_fields = [
            SearchIndexFieldReference(name="id")
        ],
    ),
)
)

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated
successfully.")
```

Source-specific properties

You can pass the following properties to create a search index knowledge source.

[Expand table](#)

Name	Description	Type	Editable	Required
name	The name of the knowledge source, which must be unique within the knowledge sources collection and follow	String	No	Yes

Name	Description	Type	Editable	Required
	the naming guidelines for objects in Azure AI Search.			
<code>description</code>	A description of the knowledge source.	String	Yes	No
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in both the knowledge source and the generated objects.	Object	Yes	No
<code>search_index_parameters</code>	Parameters specific to search index knowledge sources: <code>search_index_name</code> , <code>semantic_configuration_name</code> , <code>source_data_fields</code> , and <code>search_fields</code> .	Object	Yes	Yes
<code>search_index_name</code>	The name of the existing search index.	String	Yes	Yes
<code>semantic_configuration_name</code>	Overrides the default semantic configuration for the search index.	String	Yes	No
<code>source_data_fields</code>	The index fields returned when you specify <code>include_reference_source_data</code> in the knowledge base definition. These fields are used for citations and should be <code> retrievable</code> . Examples include the document name, file name, page numbers, or chapter numbers.	Array	Yes	No
<code>search_fields</code>	The index fields to specifically search against. When unspecified, all fields are searched.	Array	Yes	No

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
    "@odata.context": "https://my-search-
service.search.windows.net/$metadata#knowledgebases(name)",
    "value": [
        {
            "name": "my-kb"
        },
        {
            "name": "my-kb-2"
        }
    ]
}
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition
import requests
import json

endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}
```

```
response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
  "name": "my-kb",
  "description": null,
  "retrievalInstructions": null,
  "answerInstructions": null,
  "outputMode": null,
  "knowledgeSources": [
    {
      "name": "my-blob-ks",
    }
  ],
  "models": [],
  "encryptionKey": null,
  "retrievalReasoningEffort": {
    "kind": "low"
  }
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_base("knowledge_base_name")
print("Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
```

```
index_client.delete_knowledge_source("knowledge_source_name")
print(f"Knowledge source deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
 - [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
 - [Azure OpenAI demo featuring agentic retrieval](#) ↗
-

Last updated on 11/21/2025

Create a Web Knowledge Source resource

Important

- Web Knowledge Source, which uses Grounding with Bing Search and/or Grounding with Bing Custom Search, is a [First Party Consumption Service](#) governed by the [Grounding with Bing terms of use](#) and the [Microsoft Privacy Statement](#).
- The [Microsoft Data Protection Addendum](#) doesn't apply to data sent to Web Knowledge Source. When Customer uses Web Knowledge Source, Customer Data flows outside the Azure compliance and Geo boundary. This also means use of Web Knowledge Source waives all elevated Government Community Cloud security and compliance commitments to include data sovereignty and screened/citizenship-based support, as applicable.
- Use of Web Knowledge Source incurs costs; learn more about [pricing](#).
- Learn more about how Azure admins can [manage access to use of Web Knowledge Source](#).

Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Web Knowledge Source enables retrieval of real-time web data from Microsoft Bing in an agentic retrieval pipeline. [Knowledge sources](#) are created independently, referenced in a [knowledge base](#), and used as grounding data when an agent or chatbot calls a [retrieve action](#) at query time.

Bing Custom Search is always the search provider for Web Knowledge Source. Although you can't specify alternative search providers or engines, you can include or exclude specific *domains*, such as <https://learn.microsoft.com>. When no domains are specified, Web Knowledge Source has unrestricted access to the entire public internet.

Web Knowledge Source works best alongside other knowledge sources. Use Web Knowledge Source when your proprietary content doesn't provide complete, up-to-date answers or when you want to supplement results with information from a commercial search engine.

When you use Web Knowledge Source, keep the following in mind:

- The response is always a single, formulated answer to the query instead of raw search results from the web.
- Because Web Knowledge Source doesn't support extractive data, your knowledge base must use [answer synthesis](#) and [low or medium reasoning effort](#). You also can't define answer instructions.

Prerequisites

- An Azure subscription with [access to Web Knowledge Source](#). By default, access is enabled. Contact your admin if access is disabled.
- An Azure AI Search service in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#). The service must also be in an [Azure public region](#), as Web Knowledge Source isn't supported in private or sovereign clouds.
- The latest preview version of the [azure-search-documents client library](#) for Python.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#), but you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).

Check for existing knowledge sources

A knowledge source is a top-level, reusable object. Knowing about existing knowledge sources is helpful for either reuse or naming new objects.

Run the following code to list knowledge sources by name and type.

Python

```
# List knowledge sources by name and type
import requests
import json

endpoint = "{search_url}/knowledgesources"
params = {"api-version": "2025-11-01-preview", "$select": "name, kind"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge source by name to review its JSON definition.

Python

```
# Get a knowledge source definition
import requests
import json

endpoint = "{search_url}/knowledgesources/{knowledge_source_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a Web Knowledge Source resource.

JSON

```
{
  "name": "my-web-ks",
  "kind": "web",
  "description": "A sample Web Knowledge Source.",
  "encryptionKey": null,
  "webParameters": {
    "domains": null
  }
}
```

Create a knowledge source

Run the following code to create a Web Knowledge Source resource.

Python

```
# Create Web Knowledge Source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import WebKnowledgeSource,
WebKnowledgeSourceParameters, WebKnowledgeSourceDomains

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))

knowledge_source = WebKnowledgeSource(
    name = "my-web-ks",
    description = "A sample Web Knowledge Source.",
    encryption_key = None,
    web_parameters = WebKnowledgeSourceParameters(
        domains = WebKnowledgeSourceDomains(
            allowed_domains = [ { "address": "learn.microsoft.com",
"include_subpages": True } ],
            blocked_domains = [ { "address": "bing.com", "include_subpages": False } ])))
```

```

        ]
    )
}

index_client.create_or_update_knowledge_source(knowledge_source)
print(f"Knowledge source '{knowledge_source.name}' created or updated
successfully.")

```

Source-specific properties

You can pass the following properties to create a Web Knowledge Source resource.

[Expand table](#)

Name	Description	Type	Editable	Required
<code>name</code>	The name of the knowledge source, which must be unique within the knowledge sources collection and follow the naming guidelines for objects in Azure AI Search.	String	Yes	Yes
<code>description</code>	A description of the knowledge source. When unspecified, Azure AI Search applies a default description.	String	Yes	No
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in the knowledge source.	Object	Yes	No
<code>web_parameters</code>	Parameters specific to Web Knowledge Source. Currently, this is only <code>domains</code> .	Object	Yes	No
<code>domains</code>	Domains to allow or block from the search space. By default, the knowledge source uses Grounding with Bing Search to search the entire public internet. When you specify domains, the knowledge source uses Grounding with Bing Custom Search to restrict results to the specified domains. In both cases, Bing Custom Search is the search provider.	Object	Yes	No
<code>allowed_domains</code>	Domains to include in the search space. For each domain, you must specify its <code>address</code> in the <code>website.com</code> format. You can also specify whether to include the domain's subpages by setting <code>include_subpages</code> to <code>true</code> or <code>false</code> .	Array	Yes	No
<code>blocked_domains</code>	Domains to exclude from the search space. For each domain, you must specify its <code>address</code> in the <code>website.com</code> format. You can also specify whether to	Array	Yes	No

Name	Description	Type	Editable	Required
	include the domain's subpages by setting <code>include_subpages</code> to <code>true</code> or <code>false</code> .			

Assign to a knowledge base

If you're satisfied with the knowledge source, continue to the next step: specify the knowledge source in a [knowledge base](#).

After the knowledge base is configured, use the [retrieve action](#) to query the knowledge source.

Delete a knowledge source

Before you can delete a knowledge source, you must delete any knowledge base that references it or update the knowledge base definition to remove the reference. For knowledge sources that generate an index and indexer pipeline, all *generated objects* are also deleted. However, if you used an existing index to create a knowledge source, your index isn't deleted.

If you try to delete a knowledge source that's in use, the action fails and returns a list of affected knowledge bases.

To delete a knowledge source:

1. Get a list of all knowledge bases on your search service.

Python

```
# Get knowledge bases
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
    "@odata.context": "https://my-search-
service.search.windows.net/$metadata#knowledgebases(name)",
```

```
        "value": [
            {
                "name": "my-kb"
            },
            {
                "name": "my-kb-2"
            }
        ]
    }
```

2. Get an individual knowledge base definition to check for knowledge source references.

Python

```
# Get a knowledge base definition
import requests
import json

endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

An example response might look like the following:

JSON

```
{
    "name": "my-kb",
    "description": null,
    "retrievalInstructions": null,
    "answerInstructions": null,
    "outputMode": null,
    "knowledgeSources": [
        {
            "name": "my-blob-ks",
        }
    ],
    "models": [],
    "encryptionKey": null,
    "retrievalReasoningEffort": {
        "kind": "low"
    }
}
```

3. Either delete the knowledge base or [update the knowledge base](#) to remove the knowledge source if you have multiple sources. This example shows deletion.

Python

```
# Delete a knowledge base
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_base("knowledge_base_name")
print(f"Knowledge base deleted successfully.")
```

4. Delete the knowledge source.

Python

```
# Delete a knowledge source
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))
index_client.delete_knowledge_source("knowledge_source_name")
print(f"Knowledge source deleted successfully.")
```

Related content

- [Manage access to Web Knowledge Source in your Azure subscription](#)
- [Agentic retrieval in Azure AI Search](#)

Last updated on 11/21/2025

Manage access to Web Knowledge Source in your Azure subscription

ⓘ Important

- Web Knowledge Source, which uses Grounding with Bing Search and/or Grounding with Bing Custom Search, is a [First Party Consumption Service](#) governed by the [Grounding with Bing terms of use](#) and the [Microsoft Privacy Statement](#).
- The [Microsoft Data Protection Addendum](#) doesn't apply to data sent to Web Knowledge Source. When Customer uses Web Knowledge Source, Customer Data flows outside the Azure compliance and Geo boundary. This also means use of Web Knowledge Source waives all elevated Government Community Cloud security and compliance commitments to include data sovereignty and screened/citizenship-based support, as applicable.
- Use of Web Knowledge Source incurs costs; learn more about [pricing](#).

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

As an Azure admin, you can use the Azure CLI to enable or disable the use of [Web Knowledge Source](#) at the subscription level. This setting applies to all search services within the specified subscription.

Prerequisites

- Have **Owner** or **Contributor** access to the subscription.
- Have the [Azure CLI](#) installed. If you're not already signed in to Azure, run `az login`.

Check the current access state

To check the current status of Web Knowledge Source access, run the following command.

PowerShell

Azure CLI

```
az feature show --name WebKnowledgeSourceDisabled --namespace Microsoft.Search -  
-subscription "<subscription-id>"
```

The output shows the `state` property, which indicates the current registration status:

- `Registered` means access is **disabled**.
- `Unregistered` means access is **enabled**, which is the default state.

Enable use of Web Knowledge Source

Access to Web Knowledge Source is enabled by default. If access has been disabled, you can run the following command to enable it.

PowerShell

PowerShell

```
az feature unregister --name WebKnowledgeSourceDisabled --namespace  
Microsoft.Search --subscription "<subscription-id>"
```

Disable use of Web Knowledge Source

Run the following command to disable access to Web Knowledge Source.

PowerShell

PowerShell

```
az feature register --name WebKnowledgeSourceDisabled --namespace  
Microsoft.Search --subscription "<subscription-id>"
```

Related content

- Create a Web Knowledge Source resource
 - Agentic retrieval in Azure AI Search
-

Last updated on 11/19/2025

Create a knowledge base in Azure AI Search

⚠ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In Azure AI Search, a *knowledge base* is a top-level object that orchestrates [agentic retrieval](#). It defines which knowledge sources to query and the default behavior for retrieval operations. At query time, the [retrieve method](#) targets the knowledge base to run the configured retrieval pipeline.

A knowledge base specifies:

- One or more knowledge sources that point to searchable content.
- An optional LLM that provides reasoning capabilities for query planning and answer formulation.
- A retrieval reasoning effort that determines whether an LLM is invoked and manages cost, latency, and quality.
- Custom properties that control routing, source selection, output format, and object encryption.

After you create a knowledge base, you can update its properties at any time. If the knowledge base is in use, updates take effect on the next retrieval.

ⓘ Important

2025-11-01-preview renames the 2025-08-01-preview *knowledge agent* to *knowledge base*. This is a breaking change. We recommend [migrating existing code](#) to the new APIs as soon as possible.

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#). You must have [semantic ranker enabled](#). If you're using a [managed identity](#) for role-based access to deployed models, your search service must be on the Basic pricing tier or higher.

- Azure OpenAI with a [supported LLM](#) deployment.
- One or more [knowledge sources](#) on your search service.
- Permission to create and use objects on Azure AI Search. We recommend [role-based access](#). **Search Service Contributor** can create and manage a knowledge base. **Search Index Data Reader** can run queries. Alternatively, you can use [API keys](#) if a role assignment isn't feasible. For more information, see [Connect to a search service](#).
- The latest preview version of the [azure-search-documents client library](#) for Python.

 **Note**

Although you can use the Azure portal to create knowledge bases, the portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Supported models

Use one of the following LLMs from Azure OpenAI or an equivalent open-source model. For deployment instructions, see [Deploy Azure OpenAI models with Microsoft Foundry](#).

- `gpt-4o`
- `gpt-4o-mini`
- `gpt-4.1`
- `gpt-4.1-nano`
- `gpt-4.1-mini`
- `gpt-5`
- `gpt-5-nano`
- `gpt-5-mini`

Configure access

Azure AI Search needs access to the LLM from Azure OpenAI. We recommend Microsoft Entra ID for authentication and role-based access for authorization. You must be an **Owner** or **User Access Administrator** to assign roles. If roles aren't feasible, use key-based authentication instead.

[Use roles](#)

1. Configure Azure AI Search to use a managed identity.
2. On your model provider, such as Foundry Models, assign **Cognitive Services User** to the managed identity of your search service. If you're testing locally, assign the same role to your user account.
3. For local testing, follow the steps in [Quickstart: Connect without keys](#) to sign in to a specific subscription and tenant. Use `DefaultAzureCredential` instead of `AzureKeyCredential` in each request, which should look similar to the following example:

Python

```
# Authenticate using roles
from azure.identity import DefaultAzureCredential
index_client = SearchIndexClient(endpoint = "search_url", credential =
DefaultAzureCredential())
```

Important

Code snippets in this article use API keys. If you use role-based authentication, update each request accordingly. In a request that specifies both approaches, the API key takes precedence.

Check for existing knowledge bases

Knowing about existing knowledge bases is helpful for either reuse or naming new objects. Any 2025-08-01-preview knowledge agents are returned in the knowledge bases collection.

Run the following code to list existing knowledge bases by name.

Python

```
# List knowledge bases by name
import requests
import json

endpoint = "{search_url}/knowledgebases"
params = {"api-version": "2025-11-01-preview", "$select": "name"}
headers = {"api-key": "{api_key}"}
```

```
response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

You can also return a single knowledge base by name to review its JSON definition.

Python

```
# Get a knowledge base definition
import requests
import json

endpoint = "{search_url}/knowledgebases/{knowledge_base_name}"
params = {"api-version": "2025-11-01-preview"}
headers = {"api-key": "{api_key}"}

response = requests.get(endpoint, params = params, headers = headers)
print(json.dumps(response.json(), indent = 2))
```

The following JSON is an example response for a knowledge base.

JSON

```
{
  "name": "my-kb",
  "description": "A sample knowledge base.",
  "retrievalInstructions": null,
  "answerInstructions": null,
  "outputMode": null,
  "knowledgeSources": [
    {
      "name": "my-blob-ks"
    }
  ],
  "models": [],
  "encryptionKey": null,
  "retrievalReasoningEffort": {
    "kind": "low"
  }
}
```

Create a knowledge base

A knowledge base drives the agentic retrieval pipeline. In application code, it's called by other agents or chatbots.

A knowledge base connects knowledge sources (searchable content) to an LLM deployment from Azure OpenAI. Properties on the LLM establish the connection, while properties on the knowledge source establish defaults that inform query execution and the response.

Run the following code to create a knowledge base.

Python

```
# Create a knowledge base
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import KnowledgeBase,
KnowledgeBaseAzureOpenAIModel, KnowledgeSourceReference,
AzureOpenAIVectorizerParameters, KnowledgeRetrievalOutputMode,
KnowledgeRetrievalLowReasoningEffort

index_client = SearchIndexClient(endpoint = "search_url", credential =
AzureKeyCredential("api_key"))

aoai_params = AzureOpenAIVectorizerParameters(
    resource_url = "aoai_endpoint",
    deployment_name = "aoai_gpt_deployment",
    model_name = "aoai_gpt_model",
)

knowledge_base = KnowledgeBase(
    name = "my-kb",
    description = "This knowledge base handles questions directed at two unrelated sample indexes.",
    retrieval_instructions = "Use the hotels knowledge source for queries about where to stay, otherwise use the earth at night knowledge source.",
    answer_instructions = "Provide a two sentence concise and informative answer based on the retrieved documents.",
    output_mode = KnowledgeRetrievalOutputMode.ANSWER_SYNTHESIS,
    knowledge_sources = [
        KnowledgeSourceReference(name = "hotels-ks"),
        KnowledgeSourceReference(name = "earth-at-night-ks"),
    ],
    models = [KnowledgeBaseAzureOpenAIModel(azure_open_ai_parameters =
aoai_params)],
    encryption_key = None,
    retrieval_reasoning_effort = KnowledgeRetrievalLowReasoningEffort,
)

index_client.create_or_update_knowledge_base(knowledge_base)
print(f"Knowledge base '{knowledge_base.name}' created or updated successfully.")
```

Knowledge base properties

You can pass the following properties to create a knowledge base.

 Expand table

Name	Description	Type	Required
<code>name</code>	The name of the knowledge base, which must be unique within the knowledge bases collection and follow the naming guidelines for objects in Azure AI Search.	String	Yes
<code>description</code>	A description of the knowledge base. The LLM uses the description to inform query planning.	String	No
<code>retrieval_instructions</code>	A prompt for the LLM to determine whether a knowledge source should be in scope for a query, which is recommended when you have multiple knowledge sources. This field influences both knowledge source selection and query formulation. For example, instructions could append information or prioritize a knowledge source. Instructions are passed directly to the LLM, which means it's possible to provide instructions that break query planning, such as instructions that result in bypassing an essential knowledge source.	String	Yes
<code>answer_instructions</code>	Custom instructions to shape synthesized answers. The default is null. For more information, see Use answer synthesis for citation-backed responses .	String	Yes
<code>output_mode</code>	Valid values are <code>answer_synthesis</code> for an LLM-formulated answer or <code>extracted_data</code> for full search results that you can pass to an LLM as a downstream step.	String	Yes
<code>knowledge_sources</code>	One or more supported knowledge sources .	Array	Yes
<code>models</code>	A connection to a supported LLM used for answer formulation or query planning. In this preview, <code>models</code> can contain just one model, and the model provider must be Azure OpenAI. Obtain model information from the Foundry portal or a command-line request. You can use role-based access control instead of API keys for the Azure AI Search connection to the model. For more information, see How to deploy Azure OpenAI models with Foundry .	Object	No
<code>encryption_key</code>	A customer-managed key to encrypt sensitive information in both the knowledge base and the generated objects.	Object	No
<code>retrieval_reasoning_effort</code>	Determines the level of LLM-related query processing. Valid values are <code>minimal</code> , <code>low</code> (default),	Object	No

Name	Description	Type	Required
	and <code>medium</code> . For more information, see Set the retrieval reasoning effort .		

Query a knowledge base

Call the `retrieve` action on the knowledge base to verify the LLM connection and return results. For more information about the `retrieve` request and response schema, see [Retrieve data using a knowledge base in Azure AI Search](#).

Replace "Where does the ocean look green?" with a query string that's valid for your knowledge sources.

Python

```
# Send grounding request
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.knowledgebases import KnowledgeBaseRetrievalClient
from azure.search.documents.knowledgebases.models import KnowledgeBaseMessage,
KnowledgeBaseMessageTextContent, KnowledgeBaseRetrievalRequest,
RemoteSharePointKnowledgeSourceParams

kb_client = KnowledgeBaseRetrievalClient(endpoint = "search_url",
knowledge_base_name = "knowledge_base_name", credential =
AzureKeyCredential("api_key"))

request = KnowledgeBaseRetrievalRequest(
    messages=[
        KnowledgeBaseMessage(
            role = "assistant",
            content = [KnowledgeBaseMessageTextContent(text = "Use the earth at
night index to answer the question. If you can't find relevant content, say you
don't know.")]
        ),
        KnowledgeBaseMessage(
            role = "user",
            content = [KnowledgeBaseMessageTextContent(text = "Where does the ocean
look green?")]
        ),
    ],
    knowledge_source_params=[
        SearchIndexKnowledgeSourceParams(
            knowledge_source_name = "earth-at-night-ks",
            include_references = True,
            include_reference_source_data = True,
            always_query_source = False,
        )
    ],
    include_activity = True,
```

```
)  
  
result = kb_client.retrieve(request)  
print(result.response[0].content[0].text)
```

Key points:

- [messages](#) is required, but you can run this example using just the `user` role that provides the query.
- [knowledge_source_params](#) specifies one or more query targets. For each knowledge source, you can specify how much information to include in the output.

The response to the sample query might look like the following example:

HTTP

```
"response": [  
  {  
    "content": [  
      {  
        "type": "text",  
        "text": "The ocean appears green off the coast of Antarctica due to phytoplankton flourishing in the water, particularly in Granite Harbor near Antarctica's Ross Sea, where they can grow in large quantities during spring, summer, and even autumn under the right conditions [ref_id:0]. Additionally, off the coast of Namibia, the ocean can also look green due to blooms of phytoplankton and yellow-green patches of sulfur precipitating from bacteria in oxygen-depleted waters [ref_id:1]. In the Strait of Georgia, Canada, the waters turned bright green due to a massive bloom of coccolithophores, a type of phytoplankton [ref_id:5]. Furthermore, a milky green and blue bloom was observed off the coast of Patagonia, Argentina, where nutrient-rich waters from different currents converge [ref_id:6]. Lastly, a large bloom of cyanobacteria was captured in the Baltic Sea, which can also give the water a green appearance [ref_id:9]."  
      }  
    ]  
  }  
]
```

Delete a knowledge base

If you no longer need the knowledge base or need to rebuild it on your search service, use this request to delete the object.

Python

```
# Delete a knowledge base  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.indexes import SearchIndexClient
```

```
index_client = SearchIndexClient(endpoint = "search_url", credential =  
AzureKeyCredential("api_key"))  
index_client.delete_knowledge_base("knowledge_base_name")  
print(f"Knowledge base deleted successfully.")
```

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Last updated on 11/21/2025

Use answer synthesis for citation-backed responses in Azure AI Search

(!) Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

By default, a [knowledge base](#) in Azure AI Search performs *data extraction*, which returns raw grounding chunks from your knowledge sources. Data extraction is useful for retrieving specific information but lacks the context and reasoning necessary for complex queries.

You can instead enable *answer synthesis*, which uses the LLM specified in your knowledge base to answer queries in natural language. Each answer includes citations to the retrieved sources and follows any instructions you provide, such as using bulleted lists.

You can enable answer synthesis in two ways:

- On the knowledge base (becomes the default for all queries)
- On individual retrieval requests (overrides the default)

(i) Important

- The `minimal` retrieval reasoning effort disables LLM processing, so it's incompatible with answer synthesis in both knowledge base definitions and retrieval requests. For more information, see [Set the retrieval reasoning effort](#).
- Answer synthesis incurs pay-as-you-go charges from Azure OpenAI, which is based on the number of input and output tokens. Charges appear under the LLM assigned to the knowledge base. For more information, see [Availability and pricing of agentic retrieval](#).

Prerequisites

- A knowledge base that uses the [2025-11-01-preview syntax](#).

- Visual Studio Code [with the REST Client extension](#) or a prerelease package of an Azure SDK that provides the knowledge base REST APIs.

! Note

Although you can use the Azure portal to configure answer synthesis, the portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Enable answer synthesis on a knowledge base

This section explains how to enable answer synthesis on an existing knowledge base. Although you can use this configuration for new knowledge bases, knowledge base creation is beyond the scope of this article.

To enable answer synthesis on a knowledge base:

1. Use the 2025-11-01-preview of [Knowledge Base - Create or Update \(REST API\)](#) to formulate the request.
2. In the body of the request, set `outputMode` to `answerSynthesis`.
3. (Optional) Use `answerInstructions` to customize the answer output. Our example instructs the knowledge base to `Use concise bulleted lists`.

HTTP

```
@search-url = <YOUR SEARCH SERVICE URL>
@api-key = <YOUR API KEY>
@knowledge-base-name = <YOUR KNOWLEDGE BASE NAME>

### Enable answer synthesis on a knowledge base
PUT {{search-url}}/knowledgebases/{{knowledge-base-name}}?api-version=2025-11-01-
preview HTTP/1.1
Content-Type: application/json
api-key: {{api-key}}


{
  "name": "{{knowledge-base-name}}",
  "knowledgeSources": [ ... // OMITTED FOR BREVITY ],
  "models": [ ... // OMITTED FOR BREVITY ],
  "outputMode": "answerSynthesis",
  "answerInstructions": "Use concise bulleted lists"
}
```

(!) Note

This example assumes that you're using key-based authentication for local proof-of-concept testing. We recommend role-based access control for production workloads. For more information, see [Connect to Azure AI Search using roles](#).

Enable answer synthesis on a retrieval request

For per-query control over the response format, you can enable answer synthesis at query time. This approach overrides the default output mode specified in the knowledge base.

To enable answer synthesis on a retrieval request:

1. Use the 2025-11-01-preview of [Knowledge Retrieval - Retrieve \(REST API\)](#) to formulate the request.
2. In the body of the request, set `outputMode` to `answerSynthesis`.

HTTP

```
@search-url = <YOUR SEARCH SERVICE URL>
@api-key = <YOUR API KEY>
@knowledge-base-name = <YOUR KNOWLEDGE BASE NAME>

### Enable answer synthesis on a retrieval request
POST {{search-url}}/knowledgebases/{{knowledge-base-name}}/retrieve?api-
version=2025-11-01-preview HTTP/1.1
Content-Type: application/json
api-key: {{api-key}}


{
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "What is healthcare?"
        }
      ]
    }
  ],
  "outputMode": "answerSynthesis"
}
```

(!) Note

This example assumes that you're using key-based authentication for local proof-of-concept testing. We recommend role-based access control for production workloads. For more information, see [Connect to Azure AI Search using roles](#).

Get a synthesized answer

When answer synthesis is enabled, [Knowledge Retrieval - Retrieve \(REST API\)](#) returns a natural-language answer based on the instructions you optionally specified in the knowledge base. Citations to your knowledge sources are formatted as `[ref_id:<number>]`.

For example, if your instructions are `Use concise bulleted lists` and your query is `What is healthcare?`, the response might look like this:

JSON

```
{  
  "response": [  
    {  
      "content": [  
        {  
          "type": "text",  
          "text": "- Healthcare encompasses various services provided to patients  
and the general population ... // TRIMMED FOR BREVITY"  
        }  
      ]  
    }  
  ]  
}
```

The full `text` output is as follows:

```
- Healthcare encompasses various services provided to patients and the general  
population, including primary health services, hospital care, dental care, mental  
health services, and alternative health services [ref_id:1].\n- It involves the  
delivery of safe, effective, patient-centered care through different modalities,  
such as in-person encounters, shared medical appointments, and group education  
sessions [ref_id:0].\n- Behavioral health is a significant aspect of healthcare,  
focusing on the connection between behavior and overall health, including mental  
health and substance use [ref_id:2].\n- The healthcare system aims to ensure quality  
of care, access to providers, and accountability for positive outcomes while  
managing costs effectively [ref_id:2].\n- The global health system is evolving to  
address complex health needs, emphasizing the importance of cross-sectoral  
collaboration and addressing social determinants of health [ref_id:4]."
```

Depending on your knowledge base's configuration, the response might include other information, such as activity logs and reference arrays. For more information, see [Create a knowledge base](#).

Related content

- [Quickstart: Agentic retrieval in Azure AI Search \(uses answer synthesis\)](#) ↗
 - [Azure AI Search Blob knowledge source Python sample \(uses answer synthesis\)](#) ↗
 - [Agentic retrieval in Azure AI Search](#)
 - [Create a knowledge base](#)
 - [Create a search index knowledge source](#)
 - [Create a blob knowledge source](#)
-

Last updated on 11/18/2025

Set the retrieval reasoning effort

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In agentic retrieval, you can specify the level of large language model (LLM) processing for query planning and answer formulation. Use the `retrievalReasoningEffort` property to set LLM processing levels that affect costs and latency. Extra LLM processing improves relevancy, but it also takes longer and uses billable LLM resources. You can set this property in a knowledge base or on a retrieve request.

Levels of reasoning effort include:

[] Expand table

Level	Effort
<code>minimal</code>	No LLM processing. You provide the query.
<code>low</code>	Runs a single pass of LLM-based query planning and knowledge source selection. This is the default. The LLM analyzes the query and breaks it into component parts as needed.
<code>medium</code>	Adds deeper search and an enhanced retrieval stack to agentic retrieval to maximize completeness.

Prerequisites

- Azure AI Search in any [region that provides agentic retrieval](#).
- Familiarity with [agentic retrieval concepts and workflow](#).
- [A knowledge base](#) and a [knowledge source](#).
- [Visual Studio Code](#) with the [REST Client extension](#). You can also use a preview package of an Azure SDK that provides the latest knowledge source REST APIs.

Set `retrievalReasoningEffort` in a knowledge base

To establish the default behavior, set the property in the knowledge base.

1. Use [Create or Update Knowledge Base](#) to set the `retrievalReasoningEffort`.

2. Add the `retrievalReasoningEffort` property. The following JSON shows the syntax. For more information about knowledge bases, see [Create a knowledge base](#).

JSON

```
"retrievalReasoningEffort": { /* no other parameters when effort is minimal */
    "kind": "low"
}
```

Set `retrievalReasoningEffort` in a retrieve request

To override the default on a query-by-query basis, set the property in the retrieve request.

1. Modify a [retrieve action](#) to override the knowledge base `retrievalReasoningEffort` default.

2. Add the `retrievalReasoningEffort` property. A retrieve request might look similar to the following example.

JSON

```
{
  "messages": [ /* trimmed for brevity */ ],
  "retrievalReasoningEffort": { "kind": "low" },
  "outputMode": "answerSynthesis",
  "maxRuntimeInSeconds": 30,
  "maxOutputSize": 6000
}
```

Choose a retrieval reasoning effort

 Expand table

Level	Description	Recommendation	Limits
<code>minimal</code>	Disables LLM-based query planning to deliver the lowest cost and latency for agentic retrieval. It issues direct text and vector searches across the knowledge sources listed in the knowledge base, and returns the best-matching passages. Because all knowledge sources in the knowledge base are always searched and no query expansion is performed, behavior is predictable	Use "minimal" for migrations from the Search API or when you want to manage query planning yourself.	<code>outputMode</code> must be set to <code>extractiveData</code> . Answer synthesis and web knowledge aren't supported.

Level	Description	Recommendation	Limits
	and easy to control. It also means the <code>alwaysQueryKnowledgeSource</code> property on a retrieve request is ignored.		
low	The default mode of agentic retrieval, running a single pass of LLM-based query planning and knowledge source selection. The agentic retrieval engine generates subqueries and fans them out to the selected knowledge sources, then merges the results. You can enable answer synthesis to produce a grounded natural-language response with inline citations.	Use "low" when you want a balance between minimal latency and deeper processing.	5,000 answer tokens. Maximum three subqueries from a maximum of three knowledge sources. Maximum of 50 documents for semantic ranking, and 10 documents if the semantic ranker uses L3 classification.
medium	Adds deeper search and an enhanced retrieval stack to agentic retrieval to maximize completeness. After the first search is performed, a high-precision semantic classifier evaluates the retrieved documents to determine whether further processing and L3 ranking is required. If the initial results from the first pass are insufficiently relevant to the query, a follow-up iteration is performed using a revised query plan. This revised query plan takes the previous results into account and iterates by fine-tuning queries, broadening terms, or adding other knowledge sources such as the web. It also increases resource limits compared to low and minimal effort. This reasoning level optimizes for relevance rather than exhaustive recall.	Use "medium" to maximize the utility of LLM-assisted knowledge retrieval.	Medium isn't available in all agentic retrieval regions. See the list in the next section for available regions. 10,000 answer tokens. Maximum of five subqueries from a maximum of five knowledge sources. Maximum of 50 documents for semantic ranking, and 20 documents if the semantic ranker uses L3 classification.

Medium retrieval and iterative search

A medium retrieval reasoning effort provides iterative search if initial results aren't sufficiently relevant. An extra *semantic classifier model* is called to determine if a second iteration is

necessary.

The semantic classifier performs the following:

- Recognizes when there's enough context to answer the question.
- Retries on insufficient results, using existing information for context. New queries might drill down for more focused detail, or broaden the search. The activity log in the response shows the generated queries used for a more comprehensive answer.
- Rescores using L3 classification. The range is identical to L2 ranking, an absolute range of zero through 4.0.

There's only one retry. Each iteration adds latency and cost, so the system constrains retry to one pass. A second iteration adds input tokens to the query pipeline, which adds to the overall billable input token count.

Iteration can reuse or choose different sources. The second pass selects the most promising knowledge resource to provide the missing information.

Regions supporting medium retrieval reasoning effort

You can set a medium retrieval reasoning effort if your search service is in one of the following regions.

- East US 2
- East US
- South Central US
- West US 3
- West US 2
- West US
- Germany West Central
- North Europe
- Switzerland North
- Sweden Central
- Spain Central
- UK South
- Korea Central
- Japan East
- Southeast Asia

Related content

- Agentic retrieval in Azure AI Search
 - Agentic RAG: Build a reasoning retrieval engine with Azure AI Search (YouTube video) ↗
 - Azure OpenAI demo featuring agentic retrieval ↗
-

Last updated on 12/04/2025

Retrieve data using a knowledge base in Azure AI Search

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In an agentic retrieval multi-query pipeline, query execution is through the [retrieve action](#) on a knowledge base that invokes parallel query processing. This request structure is updated for the new 2025-11-01-preview, which introduces breaking changes from previous previews. For help with breaking changes, see [Migrate your agentic retrieval code](#).

This article explains how to set up a retrieve action. It also covers the three components of the retrieval response:

- *extracted response for the LLM*
- *referenced results*
- *query activity*

A retrieve request can include instructions for query processing that override the default instructions set on the knowledge base. A retrieve action has core parameters that are supported on any request, plus parameters that are specific to a knowledge source.

Prerequisites

- A [supported knowledge source](#) that wraps a searchable index or points to an external source for native data retrieval.
- A [knowledge base](#) represents one or more knowledge sources, plus a chat completion model if you want intelligent query planning and answer formulation.
- Azure AI Search in any [region that provides agentic retrieval](#).
- Permissions on Azure AI Search. Roles for retrieving content include **Search Index Data Reader** for running queries. To support an outbound call from a search service to a chat completion model, you must configure a managed identity for the search service, and it must have **Cognitive Services User** permissions on the Azure OpenAI resource. For more

information about local testing and obtaining access tokens, see [Quickstart: Connect without keys](#).

- API version requirements. To create or use a knowledge base, use the [2025-11-01-preview](#) data plane REST API. Or, use a preview package of an Azure SDK that provides knowledge base APIs: [Python](#), [.NET](#), [Java](#).

To follow the steps in this guide, we recommend [Visual Studio Code](#) with a [REST client](#) for sending REST API calls to Azure AI Search.

 **Note**

Although you can use the Azure portal to retrieve data from knowledge bases, the portal uses the 2025-08-01-preview, which uses the previous "knowledge agent" terminology and doesn't support all 2025-11-01-preview features. For help with breaking changes, see [Migrate your agentic retrieval code](#).

Set up the retrieve action

A retrieve action is specified on a [knowledge base](#). The knowledge base has one or more knowledge sources. Retrieval can return a synthesized answer in natural language or raw grounding chunks from the knowledge sources.

- Review your knowledge base definition to understand which knowledge sources are in scope.
- Review your knowledge sources to understand their parameters and configuration.
- Use the [2025-11-01-preview](#) data plane REST API or an Azure SDK preview package to call retrieve.

For knowledge sources that have default retrieval instructions, you can override the defaults in the retrieve request.

Retrieval from a search index

For knowledge sources that target a search index, all `searchable` fields are in-scope for query execution. If the index includes vector fields, your index should have a valid [vectorizer definition](#) so that the agentic retrieval engine can vectorize the query inputs. Otherwise, vector fields are ignored. The implied query type is `semantic`, and there's no search mode.

The input for the retrieval route is chat conversation history in natural language, where the `messages` array contains the conversation. Messages are only supported if the [retrieval reasoning effort](#) is either low or medium.

HTTP

```
@search-url=<YOUR SEARCH SERVICE URL>
@accessToken=<YOUR PERSONAL ID>

# Send grounding request
POST https://{{search-url}}/knowledgebases/{{knowledge-base-name}}/retrieve?api-
version=2025-11-01-preview
Content-Type: application/json
Authorization: Bearer {{accessToken}}


{
  "messages" : [
    {
      "role" : "assistant",
      "content" : [
        { "type" : "text", "text" : "You can answer questions about the
Earth at night.
Sources have a JSON format with a ref_id that must be cited in
the answer.
If you do not have the answer, respond with 'I do not know'." }
      ]
    },
    {
      "role" : "user",
      "content" : [
        { "type" : "text", "text" : "Why is the Phoenix nighttime street
grid is so sharply visible from space, whereas large stretches of the interstate
between midwestern cities remain comparatively dim?" }
      ]
    }
  ],
  "knowledgeSourceParams": [
    {
      "filterAddOn": null,
      "knowledgeSourceName": "earth-at-night-blob-ks",
      "kind": "searchIndex"
    }
  ]
}
```

Responses

Successful retrieval returns a `200 OK` status code. If the knowledge base fails to retrieve from one or more knowledge sources, a `206 Partial Content` status code is returned, and the

response only includes results from sources that succeeded. Details about the partial response appear as [errors in the activity array](#).

Retrieve parameters

 [Expand table](#)

Name	Description	Type	Editable	Required
<code>messages</code>	Articulates the messages sent to a chat completion model. The message format is similar to Azure OpenAI APIs.	Object	Yes	No
<code>role</code>	Defines where the message came from, for example either <code>assistant</code> or <code>user</code> . The model you use determines which roles are valid.	String	Yes	No
<code>content</code>	The message or prompt sent to the LLM. It must be text in this preview.	String	Yes	No
<code>knowledgeSourceParams</code>	Specifies parameters for each knowledge source if you want to customize the query or response at query time.	Object	Yes	No

Examples

Retrieve requests vary depending on the knowledge sources and whether you want to override a default configuration. Here are several examples that illustrate a range of requests.

Example: Override default reasoning effort and set request limits

This example specifies [answer formulation](#), so `retrievalReasoningEffort` must be "low" or "medium".

HTTP

```
POST {{url}}/knowledgebases/kb-override/retrieve?api-version={{api-version}}
api-key: {{key}}
Content-Type: application/json
```

```
{
  "messages": [
    {
```

```

        "role": "user",
        "content": [
            { "type": "text", "text": "What companies are in the financial
sector?" }
        ]
    },
    "retrievalReasoningEffort": { "kind": "low" },
    "outputMode": "answerSynthesis",
    "maxRuntimeInSeconds": 30,
    "maxOutputSize": 6000
}

```

Example: Set references for each knowledge source

This example uses the default reasoning effort specified in the knowledge base. The focus of this example is specification of how much information to include in the response.

HTTP

```

POST {{url}}/knowledgebases/kb-medium-example/retrieve?api-version={{api-version}}
api-key: {{key}}
Content-Type: application/json

{
    "messages": [
        {
            "role": "user",
            "content": [
                { "type": "text", "text": "What companies are in the financial
sector?" }
            ]
        }
    ],
    "includeActivity": true,
    "knowledgeSourceParams": [
        {
            "knowledgeSourceName": "demo-financials-ks",
            "kind": "searchIndex",
            "includeReferences": true,
            "includeReferenceSourceData": true
        },
        {
            "knowledgeSourceName": "demo-communicationservices-ks",
            "kind": "searchIndex",
            "includeReferences": false,
            "includeReferenceSourceData": false
        },
        {
            "knowledgeSourceName": "demo-healthcare-ks",
            "kind": "searchIndex",
            "includeReferences": true,

```

```
        "includeReferenceSourceData": false,
        "alwaysQuerySource": true
    }
]
}
```

⚠ Note

If you're retrieving content from a OneLake or indexed SharePoint knowledge source, set `includeReferenceSourceData` to `true` to include the source document URL in the citation.

Example: minimal reasoning effort

In this example, there's no chat completion model for intelligent query planning or answer formulation. The query string is passed to the agentic retrieval engine for keyword search or hybrid search.

HTTP

```
POST {{url}}/knowledgebases/kb-minimal/retrieve?api-version={{api-version}}
api-key: {{key}}
Content-Type: application/json

{
    "intents": [
        {
            "type": "semantic",
            "search": "what is a brokerage"
        }
    ]
}
```

Review the extracted response

The *extracted response* is single unified string that's typically passed to an LLM that consumes it as grounding data, using it to formulate a response. Your API call to the LLM includes the unified string and instructions for model, such as whether to use the grounding exclusively or as a supplement.

The body of the response is also structured in the chat message style format. Currently in this preview release, the content is serialized JSON.

HTTP

```
"response": [
  {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "[{\\"ref_id\\":0,\\\"title\\\":\\\"Urban
Structure\\\",\\\"terms\\\":\\\"Location of Phoenix, Grid of City Blocks, Phoenix
Metropolitan Area at Night\\\",\\\"content\\\":\\\"<content chunk redacted>\\\"}]"
      }
    ]
  }
]
```

Key points:

- `content.text` is a JSON array. It's a single string composed of the most relevant documents (or chunks) found in the search index, given the query and chat history inputs. This array is your grounding data that a chat completion model uses to formulate a response to the user's question.

This portion of the response consists of the 200 chunks or less, excluding any results that fail to meet the minimum threshold of a 2.5 reranker score.

The string starts with the reference ID of the chunk (used for citation purposes), and any fields specified in the semantic configuration of the target index. In this example, you should assume the semantic configuration in the target index has a "title" field, a "terms" field, and a "content" field.

- `content.type` has one valid value in this preview: `text`.

ⓘ Note

The `maxOutputSize` property on the [knowledge base](#) determines the length of the string.

Review the activity array

The activity array outputs the query plan, which helps you track the operations performed when executing the request. It also provides operational transparency so you can understand the billing implications and frequency of resource invocations.

The output includes the following components.

[] Expand table

Section	Description
modelQueryPlanning	For knowledge bases that use an LLM for query planning, this section reports on the token counts used for input, and the token count for the subqueries.
source-specific activity	For each knowledge source included in the query, report on elapsed time and which arguments were used in the query, including the semantic ranker. Knowledge source types include <code>searchIndex</code> , <code>azureBlob</code> , and other supported knowledge sources .
agenticReasoningEffort	For each retrieve action, you can specify the degree of LLM support. Use minimal to bypass an LLM, low for constrained LLM processing, and medium for full LLM processing.
modelAnswerSynthesis	For knowledge bases that specify answer formulation, this section reports on the token count for formulating the answer, and the token count of the answer output.

Output reports on the token consumption for agentic reasoning during retrieval at the specified [retrieval reasoning effort](#).

Output also includes the following information:

- Subqueries sent to the retrieval pipeline.
- Errors for any retrieval failures, such as inaccessible knowledge sources.

Here's an example of an activity array.

JSON

```

"activity": [
  {
    "type": "modelQueryPlanning",
    "id": 0,
    "inputTokens": 2302,
    "outputTokens": 109,
    "elapsedMs": 2396
  },
  {
    "type": "searchIndex",
    "id": 1,
    "knowledgeSourceName": "demo-financials-ks",
    "queryTime": "2025-11-04T19:25:23.683Z",
    "count": 26,
    "elapsedMs": 1137,
    "searchIndexArguments": {
      "search": "List of companies in the financial sector according to SEC GICS classification",
      "filter": null,
      "sourceDataFields": [ ],
      "searchFields": [ ]
    }
  }
]

```

```

        "semanticConfigurationName": "en-semantic-config"
    }
},
{
    "type": "searchIndex",
    "id": 2,
    "knowledgeSourceName": "demo-healthcare-ks",
    "queryTime": "2025-11-04T19:25:24.186Z",
    "count": 17,
    "elapsedMs": 494,
    "searchIndexArguments": {
        "search": "List of companies in the financial sector according to SEC GICS classification",
        "filter": null,
        "sourceDataFields": [ ],
        "searchFields": [ ],
        "semanticConfigurationName": "en-semantic-config"
    }
},
{
    "type": "agenticReasoning",
    "id": 3,
    "retrievalReasoningEffort": {
        "kind": "low"
    },
    "reasoningTokens": 103368
},
{
    "type": "modelAnswerSynthesis",
    "id": 4,
    "inputTokens": 5821,
    "outputTokens": 344,
    "elapsedMs": 3837
}
]

```

Review the references array

The `references` array is a direct reference from the underlying grounding data and includes the `sourceData` used to generate the response. It consists of every single document that was found and semantically ranked by the agentic retrieval engine. Fields in the `sourceData` include an `id` and semantic fields: `title`, `terms`, `content`.

The `id` is a reference ID for an item within a specific response. It's not the document key in the search index. It's used for providing citations.

The purpose of this array is to provide a chat message style structure for easy integration. For example, if you want to serialize the results into a different structure or you require some programmatic manipulation of the data before you returned it to the user.

You can also get the structured data from the source data object in the references array to manipulate it however you see fit.

Here's an example of the references array.

JSON

```
"references": [
  {
    "type": "AzureSearchDoc",
    "id": "0",
    "activitySource": 2,
    "docKey": "earth_at_night_508_page_104_verbalized",
    "sourceData": null
  },
  {
    "type": "AzureSearchDoc",
    "id": "1",
    "activitySource": 2,
    "docKey": "earth_at_night_508_page_105_verbalized",
    "sourceData": null
  }
]
```

ⓘ Note

If you're retrieving content from a OneLake or indexed SharePoint knowledge source, set `includeReferenceSourceData` to `true` on the retrieve request to get the source document URL in the citation.

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Agentic RAG: Build a reasoning retrieval engine with Azure AI Search \(YouTube video\)](#) ↗
- [Azure OpenAI demo featuring agentic retrieval](#) ↗

Migrate agentic retrieval code to the latest version

Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

If you wrote [agentic retrieval](#) code using an early preview REST API, this article explains when and how to migrate to a newer version. It also describes breaking and nonbreaking changes for all REST API versions that support agentic retrieval.

Migration instructions are intended to help you run an existing solution on a newer API version. The instructions in this article help you address breaking changes at the API level so that your app runs as before. For help with adding new functionality, start with [What's new](#).

Tip

Using Azure SDKs instead of REST? Read this article to learn about breaking changes, and then install a newer preview package to begin your updates. Before you start, check the SDK change logs to confirm API updates: [Python](#), [.NET](#), [JavaScript](#), [Java](#).

When to migrate

Each new API version that supports agentic retrieval has introduced breaking changes, from the original [2025-05-01-preview](#) to [2025-08-01-preview](#), to the latest [2025-11-01-preview](#).

You can continue to run older code with no updates if you retain the API version value. However, to benefit from bug fixes, improvements, and newer functionality, you must update your code.

How to migrate

- The supported migration path is incremental. If your code targets 2025-05-01-preview, first migrate to 2025-08-01-preview, and then migrate to 2025-11-01-preview.

- To understand the scope of changes, review [breaking and nonbreaking changes](#) for each version.
- "Migration" means creating new, uniquely named objects that implement the behaviors of the previous version. You can't overwrite an existing object if properties are added or deleted on the API. One advantage of creating new objects is the ability to preserve existing objects while new ones are developed and tested.
- For each object that you migrate, start by getting the current definition from the search service so that you can review existing properties before specifying the new one.
- Delete older versions only after your migration is fully tested and deployed.

2025-11-01-preview

If you're migrating from [2025-08-01-preview](#), *knowledge agent* is renamed to *knowledge base*, and multiple properties are relocated to different objects and levels within an object definition.

1. [Update searchIndex knowledge sources.](#)
2. [Update azureBlob knowledge sources.](#)
3. Replace knowledge agent with knowledge base.
4. Update the retrieval request and send a query to test your updates.
5. [Update client code.](#)

Update a searchIndex knowledge source

This procedure creates a new 2025-11-01-preview `searchIndex` knowledge source at the same functional level as the previous 2025-08-01 version. The underlying index itself requires no updates.

1. List all knowledge sources by name to find your knowledge source.

HTTP

```
### List all knowledge sources by name
GET {{search-endpoint}}/knowledge-sources?api-version=2025-08-01-
preview&$select=name
api-key: {{api-key}}
Content-Type: application/json
```

2. [Get the current definition](#) to review existing properties.

HTTP

```
### Get a specific knowledge source
GET {{search-endpoint}}/knowledge-sources/search-index-ks?api-version=2025-08-01-preview
api-key: {{api-key}}
Content-Type: application/json
```

The response should look similar to the following example.

JSON

```
{  
    "name": "search-index-ks",  
    "kind": "searchIndex",  
    "description": "This knowledge source pulls from a search index created using the 2025-08-01-preview.",  
    "encryptionKey": null,  
    "searchIndexParameters": {  
        "searchIndexName": "earth-at-night-idx",  
        "sourceDataSelect": "id, page_chunk, page_number"  
    },  
    "azureBlobParameters": null  
}
```

3. Formulate a [Create Knowledge Source](#) request as the basis for your migration.

Start with the 08-01-preview JSON.

HTTP

```
POST {{url}}/knowledge-sources/search-index-ks?api-version=2025-08-01-preview
api-key: {{key}}
Content-Type: application/json

{  
    "name": "search-index-ks",  
    "kind": "searchIndex",  
    "description": "A sample search index knowledge source",  
    "encryptionKey": null,  
    "searchIndexParameters": {  
        "searchIndexName": "my-search-index",  
        "sourceDataSelect": "id, page_chunk, page_number"  
    }  
}
```

Make the following updates for a 2025-11-01-preview migration:

- Give the knowledge source a new name.
- Change the API version to 2025-11-01-preview.

- Rename `sourceDataSelect` to `sourceDataFields` and change the string to an array with name-value pairs for each retrievable field you want to query. These are the fields to return in the search results, similar to a `select` clause in a classic query.

4. Review your updates and then send the request to create the object.

HTTP

```
PUT {{url}}/knowledge-sources/search-index-ks-11-01?api-version=2025-11-01-preview
api-key: {{key}}
Content-Type: application/json

{
    "name": "search-index-ks-11-01",
    "kind": "searchIndex",
    "description": "knowledge source migrated to 2025-11-01-preview",
    "encryptionKey": null,
    "searchIndexParameters": {
        "searchIndexName": "my-search-index",
        "sourceDataFields": [
            { "name": "id" }, { "name": "page_chunk" }, { "name": "page_number" }
        ]
    }
}
```

You now have a migrated `searchIndex` knowledge source is backwards compatible with the previous version, using the correct property specifications for the 2025-11-01-preview.

The response includes the full definition of the new object. For more information about new properties available to this knowledge source type, which you can now do through updates, see [How to create a search index knowledge source](#).

Update an azureBlob knowledge source

This procedure creates a new 2025-11-01-preview `azureBlob` knowledge source at the same functional level as the previous 2025-08-01 version. It creates a new set of generated objects: data source, skillset, indexer, index.

1. List all knowledge sources by name to find your knowledge source.

HTTP

```
### List all knowledge sources by name
GET {{search-endpoint}}/knowledge-sources?api-version=2025-08-01-preview&$select=name
```

```
api-key: {{api-key}}
Content-Type: application/json
```

2. Get the current definition to review existing properties.

HTTP

```
### Get a specific knowledge source
GET {{search-endpoint}}/knowledge-sources/azure-blob-ks?api-version=2025-08-01-preview
api-key: {{api-key}}
Content-Type: application/json
```

The response might look similar to the following example if your workflow includes a model. Notice that a response includes the names of the generated objects. These objects are fully independent of the knowledge source and remain operational even if you update or delete their knowledge source.

JSON

```
{
  "name": "azure-blob-ks",
  "kind": "azureBlob",
  "description": "A sample azure blob knowledge source.",
  "encryptionKey": null,
  "searchIndexParameters": null,
  "azureBlobParameters": {
    "connectionString": "<redacted>",
    "containerName": "blobcontainer",
    "folderPath": null,
    "disableImageVerbalization": false,
    "identity": null,
    "embeddingModel": {
      "name": "embedding-model",
      "kind": "azureOpenAI",
      "azureOpenAIParameters": {
        "resourceUri": "<redacted>",
        "deploymentId": "text-embedding-3-large",
        "apiKey": "<redacted>",
        "modelName": "text-embedding-3-large",
        "authIdentity": null
      },
      "customWebApiParameters": null,
      "aiServicesVisionParameters": null,
      "amlParameters": null
    },
    "chatCompletionModel": {
      "kind": "azureOpenAI",
      "azureOpenAIParameters": {
        "resourceUri": "<redacted>",
        "deploymentId": "gpt-4o-mini",
        "modelType": "chat"
      }
    }
}
```

```

        "apiKey": "<redacted>",
        "modelName": "gpt-4o-mini",
        "authIdentity": null
    }
},
"ingestionSchedule": null,
"createdResources": {
    "datasource": "azure-blob-ks-datasource",
    "indexer": "azure-blob-ks-indexer",
    "skillset": "azure-blob-ks-skillset",
    "index": "azure-blob-ks-index"
}
}
}

```

3. Formulate a [Create Knowledge Source](#) request as the basis for your migration.

Start with the 08-01-preview JSON.

HTTP

```

POST {{url}}/knowledge-sources/azure-blob-ks?api-version=2025-08-01-preview
api-key: {{key}}
Content-Type: application/json

{
    "name": "azure-blob-ks",
    "kind": "azureBlob",
    "description": "A sample azure blob knowledge source.",
    "encryptionKey": null,
    "azureBlobParameters": {
        "connectionString": "<redacted>",
        "containerName": "blobcontainer",
        "folderPath": null,
        "disableImageVerbalization": false,
        "identity": null,
        "embeddingModel": {
            "name": "embedding-model",
            "kind": "azureOpenAI",
            "azureOpenAIParameters": {
                "resourceUri": "<redacted>",
                "deploymentId": "text-embedding-3-large",
                "apiKey": "<redacted>",
                "modelName": "text-embedding-3-large",
                "authIdentity": null
            },
            "customWebApiParameters": null,
            "aiServicesVisionParameters": null,
            "amlParameters": null
        },
        "chatCompletionModel": null,
        "ingestionSchedule": null
    }
}
```

```
}
```

Make the following updates for a 2025-11-01-preview migration:

- Give the knowledge source a new name.
- Change the API version to `2025-11-01-preview`.
- Add `ingestionParameters` as a container for the following child properties:
`"embeddingModel"`, `"chatCompletionModel"`, `"ingestionSchedule"`,
`"contentExtractionMode"`.

4. Review your updates and then send the request to create the object. New generated objects are created for the indexer pipeline.

HTTP

```
PUT {{url}}/knowledge-sources/azure-blob-ks-11-01?api-version=2025-11-01-
preview
api-key: {{key}}
Content-Type: application/json

{
    "name": "azure-blob-ks",
    "kind": "azureBlob",
    "description": "A sample azure blob knowledge source",
    "encryptionKey": null,
    "azureBlobParameters": {
        "connectionString": "{{blob-connection-string}}",
        "containerName": "blobcontainer",
        "folderPath": null,
        "ingestionParameters": {
            "embeddingModel": {
                "kind": "azureOpenAI",
                "azureOpenAIParameters": {
                    "deploymentId": "text-embedding-3-large",
                    "modelName": "text-embedding-3-large",
                    "resourceUri": "{{aoai-endpoint}}",
                    "apiKey": "{{aoai-key}}"
                }
            },
            "chatCompletionModel": null,
            "disableImageVerbalization": false,
            "ingestionSchedule": null,
            "contentExtractionMode": "minimal"
        }
    }
}
```

You now have a migrated `azureBlob` knowledge source that is backwards compatible with the previous version, using the correct property specifications for the 2025-11-01-preview.

The response includes the full definition of the new object. For more information about new properties available to this knowledge source type, which you can now do through updates, see [How to create an Azure Blob knowledge source](#).

Replace knowledge agent with knowledge base

1. Knowledge bases require a knowledge source. Make sure you have a knowledge source that targets 2025-11-01-preview before you start.
2. [Get the current definition](#) to review existing properties.

HTTP

```
### Get a knowledge agent by name
GET {{search-endpoint}}/agents/earth-at-night?api-version=2025-08-01-
preview
api-key: {{api-key}}
Content-Type: application/json
```

The response might look similar to the following example.

JSON

```
{
  "name": "earth-at-night",
  "description": "A sample knowledge agent that retrieves from the earth-
at-night knowledge source.",
  "retrievalInstructions": null,
  "requestLimits": null,
  "encryptionKey": null,
  "knowledgeSources": [
    {
      "name": "earth-at-night",
      "alwaysQuerySource": null,
      "includeReferences": null,
      "includeReferenceSourceData": null,
      "maxSubQueries": null,
      "rerankerThreshold": 2.5
    }
  ],
  "models": [
    {
      "kind": "azureOpenAI",
      "azureOpenAIParameters": {
        "resourceUri": "<redacted>",
        "deploymentId": "gpt-5-mini",
        "temperature": 0.5
      }
    }
  ]
}
```

```

        "apiKey": "<redacted>",
        "modelName": "gpt-5-mini",
        "authIdentity": null
    }
},
],
"outputConfiguration": {
    "modality": "answerSynthesis",
    "answerInstructions": null,
    "attemptFastPath": false,
    "includeActivity": null
}
}

```

3. Formulate a [Create Knowledge Base](#) request as the basis for your migration.

Start with the 08-01-preview JSON.

HTTP

```

PUT {{url}}/knowledgebases/earth-at-night?api-version=2025-08-01-preview
HTTP/1.1
api-key: {{key}}
Content-Type: application/json

{
    "name": "earth-at-night",
    "description": "A sample knowledge agent that retrieves from the earth-at-night knowledge source.",
    "retrievalInstructions": null,
    "encryptionKey": null,
    "knowledgeSources": [
        {
            "name": "earth-at-night",
            "alwaysQuerySource": null,
            "includeReferences": null,
            "includeReferenceSourceData": null,
            "maxSubQueries": null,
            "rerankerThreshold": 2.5
        }
    ],
    "models": [
        {
            "kind": "azureOpenAI",
            "azureOpenAIParameters": {
                "resourceUri": "<redacted>",
                "apiKey": "<redacted>",
                "deploymentId": "gpt-5-mini",
                "modelName": "gpt-5-mini"
            }
        }
    ],
    "outputConfiguration": {

```

```
        "modality": "answerSynthesis"
    }
}
```

Make the following updates for a 2025-11-01-preview migration:

- Replace the endpoint: `/knowledgebases/{{your-object-name}}`. Give the knowledge base a unique name.
 - Change the API version to `2025-11-01-preview`.
 - Delete `requestLimits`. The `maxRuntimeInSeconds` and `maxOutputSize` properties are now specified on the retrieval request object directly
 - Update `knowledgeSources`:
 - Delete `maxSubQueries` and replace with a `retrievalReasoningEffort`` (see [Set the retrieval reasoning effort](#)).
 - Move `alwaysQuerySource`, `includeReferenceSourceData`, `includeReferences`, and `rerankerThreshold` to the `knowledgeSourcesParams` section of a [retrieve action](#).
 - No changes for `models`.
 - Update `outputConfiguration`:
 - Replace `outputConfiguration` with `outputMode`.
 - Delete `attemptFastPath`. It no longer exists. Equivalent behavior is implemented through `retrievalReasoningEffort` set to minimum (see [Set the retrieval reasoning effort](#)).
 - If modality is set to `answerSynthesis`, make sure you set the retrieval reasoning effort to low (default) or medium.
 - Add `ingestionParameters` as a requirement for creating a 2025-11-01-preview `azureBlob` knowledge source.
4. Review your updates and then send the request to create the object. New generated objects are created for the indexer pipeline.

HTTP

```
PUT {{url}}/knowledgebases/earth-at-night-11-01?api-version={{api-version}}
```

```
api-key: {{key}}
Content-Type: application/json

{
    "name": "earth-at-night-11-01",
    "description": "A sample knowledge base at the same functional level as the previous knowledge agent.",
    "retrievalInstructions": null,
    "encryptionKey": null,
    "knowledgeSources": [
        {
            "name": "earth-at-night-ks"
        }
    ],
    "models": [
        {
            "kind": "azureOpenAI",
            "azureOpenAIParameters": {
                "resourceUri": "<redacted>",
                "apiKey": "<redacted>",
                "deploymentId": "gpt-5-mini",
                "modelName": "gpt-5-mini"
            }
        }
    ],
    "retrievalReasoningEffort": null,
    "outputMode": "answerSynthesis",
    "answerInstructions": "Provide a concise and accurate answer based on the retrieved information."
}

}
```

You now have a knowledge base instead of a knowledge agent, and the object is backwards compatible with the previous version

The response includes the full definition of the new object. For more information about new properties available to a knowledge base, which you can now do through updates, see [How to create a knowledge base](#).

Update and test the retrieval for 2025-11-01-preview updates

The retrieval request is modified for the 2025-11-01-preview to support more shapes, including a simpler request that minimizes LLM processing. For more information about retrieval in this preview, see [Retrieve data using a knowledge base](#). This section explains how to update your code.

1. Change the `/agents/retrieve` endpoint to `/knowledgebases/retrieve`.
2. Change the API version to `2025-11-01-preview`.

3. No changes to `messages` are required if you are using a `low` or `medium` `retrievalReasoningEffort`. Replace `messages` with `intent` if you use `minimal` reasoning (see [Set the retrieval reasoning effort](#)).
4. Modify `knowledgeSourceParams` to include any properties that were removed from the agent: `rerankerThreshold`, `alwaysQuerySource`, `includeReferenceSourceData`, `includeReferences`.
5. Add `retrievalReasoningEffort` set to `minimum` if you were using `attemptFastPath`. If you were using `maxSubQueries`, it no longer exists. Use the `retrievalReasoningEffort` setting to specify subquery processing (see [Set the retrieval reasoning effort](#)).

To test your knowledge base's output with a query, use the 2025-11-01-preview of [Knowledge Retrieval - Retrieve \(REST API\)](#).

HTTP

```
### Send a query to the knowledge base
POST {{url}}/knowledgebases/earth-at-night-11-01/retrieve?api-version=2025-11-01-preview
api-key: {{key}}
Content-Type: application/json

{
  "messages": [
    {
      "role": "user",
      "content": [
        { "type": "text", "text": "What are some light sources on the ocean at night" }
      ]
    }
  ],
  "includeActivity": true,
  "retrievalReasoningEffort": { "kind": "medium" },
  "outputMode": "answerSynthesis",
  "maxRuntimeInSeconds": 30,
  "maxOutputSize": 6000
}
```

If the response has a `200 OK` HTTP code, your knowledge base successfully retrieved content from the knowledge source.

Update code and clients for 2025-11-01-preview

To complete your migration, follow these cleanup steps:

1. For Azure Blob knowledge sources only, update clients to use the new index. If you have code or script that runs an indexer or references a data source, index, or skillset, make sure you update the references to the new objects.
2. Replace all agent references with `knowledgeBases` in configuration files, code, scripts, and tests.
3. Update client calls to use the 2025-11-01-preview.
4. Clear or regenerate cached definitions that were created using the old shapes.

Version-specific changes

This section covers breaking and nonbreaking changes for the following REST API versions:

- [2025-11-01-preview](#)
- [2025-08-01-preview](#)
- [2025-05-01-preview](#)

2025-11-01-preview

To review the [REST API reference documentation](#) for this version, make sure the 2025-11-01-preview API version is selected in the filter at the top of the page.

Breaking changes

- Knowledge agent is renamed to knowledge base.

 [Expand table](#)

Previous Route	New Route
<code>/agents</code>	<code>/knowledgebases</code>
<code>/agents/agent-name</code>	<code>/knowledgebases/knowledge-base-name</code>
<code>/agents/agent-name/retrieve</code>	<code>/knowledgebases/knowledge-base-name/retrieve</code>

- Knowledge agent (base) `outputConfiguration` is renamed to `outputMode` and changed from an object to a string enumerator. Several properties are impacted:

- `includeActivity` is moved from `outputConfiguration` onto the retrieval request object directly.
- `attemptFastPath` in `outputConfiguration` is removed entirely. The new `minimal` reasoning effort is the replacement.
- Knowledge agent (base) `requestLimits` is removed. Its child properties of `maxRuntimeInSeconds` and `maxOutputSize` are moved onto the retrieval request object directly.
- Knowledge agent (base) `knowledgeSources` parameters now only list the names of knowledge source used by a knowledge base. Other child properties that used to be under `knowledgeSources` are moved to the `knowledgeSourceParams` properties of the retrieval request object:
 - `rerankerThreshold`
 - `alwaysQuerySource`
 - `includeReferenceSourceData`
 - `includeReferences`

The `maxSubQueries` property is gone. Its replacement is the new retrieval reasoning effort property.

- Knowledge agent (base) retrieval request object: The `semanticReranker` activity record is replaced with the `agenticReasoning` activity record type.
- Knowledge sources for both `azureBlob` and `searchIndex`: top-level properties for `identity`, `embeddingModel`, `chatCompletionModel`, `disableImageVerbalization`, and `ingestionSchedule` are now part of an `ingestionParameters` object on the knowledge source. All knowledge sources that pull from a search index have an `ingestionParameters` object.
- For `searchIndex` knowledge sources only: `sourceDataSelect` is renamed to `sourceDataFields` and is an array that accepts `fieldName` and `fieldToSearch`.

2025-08-01-preview

To review the [REST API reference documentation](#) for this version, make sure the 2025-08-01-preview API version is selected in the filter at the top of the page.

- Introduces knowledge sources as the new way to define data sources, supporting both `searchIndex` (one or multiple indexes) and `azureBlob` kinds. For more information, see [Create a search index knowledge source](#) and [Create a blob knowledge source](#).
- Requires `knowledgeSources` instead of `targetIndexes` in agent definitions. For migration steps, see [How to migrate](#).
- Removes `defaultMaxDocsForReranker` support. This property previously existed in `targetIndexes`, but there's no replacement in `knowledgeSources`.

2025-05-01-preview

This REST API version introduces agentic retrieval and knowledge agents. Each agent definition requires a `targetIndexes` array that specifies a single index and optional properties, such as `defaultRerankerThreshold` and `defaultIncludeReferenceSourceData`.

To review the [REST API reference documentation](#) for this version, make sure the 2025-05-01-preview API version is selected in the filter at the top of the page.

Related content

- [Agentic retrieval in Azure AI Search](#)
- [Create a knowledge agent](#)
- [Create a knowledge source](#)

Last updated on 11/18/2025

Create a vector query in Azure AI Search

09/29/2025

If you have a [vector index](#) in Azure AI Search, this article explains how to:

- ✓ [Query vector fields](#)
- ✓ [Query multiple vector fields at once](#)
- ✓ [Set vector weights](#)
- ✓ [Query with integrated vectorization](#)
- ✓ [Set thresholds to exclude low-scoring results \(preview\)](#)

This article uses REST for illustration. After you understand the basic workflow, continue with the Azure SDK code samples in the [azure-search-vector-samples](#) repo, which provides end-to-end solutions that include vector queries.

You can also use [Search Explorer](#) in the Azure portal.

Prerequisites

- An [Azure AI Search service](#) in any region and on any tier.
- A [vector index](#). Check for a `vectorSearch` section in your index to confirm its presence.
- Optionally, [add a vectorizer](#) to your index for built-in text-to-vector or image-to-vector conversion during queries.
- Visual Studio Code with a [REST client](#) and sample data if you want to run these examples on your own. To get started with the REST client, see [Quickstart: Full-text search using REST](#).

Convert a query string input into a vector

To query a vector field, the query itself must be a vector.

One approach for converting a user's text query string into its vector representation is to call an embedding library or API in your application code. As a best practice, *always use the same embedding models used to generate embeddings in the source documents*. You can find code samples showing [how to generate embeddings](#) in the [azure-search-vector-samples](#) repo.

A second approach is to [use integrated vectorization](#), now generally available, to have Azure AI Search handle your query vectorization inputs and outputs.

Here's a REST API example of a query string submitted to a deployment of an Azure OpenAI embedding model:

HTTP

```
POST https://{{openai-service-name}}.openai.azure.com/openai/deployments/{{openai-deployment-name}}/embeddings?api-version={{openai-api-version}}
Content-Type: application/json
api-key: {{admin-api-key}}
{
    "input": "what azure services support generative AI"
}
```

The expected response is 202 for a successful call to the deployed model.

The `embedding` field in the body of the response is the vector representation of the query string `input`. For testing purposes, you would copy the value of the `embedding` array into `vectorQueries.vector` in a query request, using the syntax shown in the next several sections.

The actual response to this POST call to the deployed model includes 1,536 embeddings. For readability, this example only shows the first few vectors.

JSON

```
{
    "object": "list",
    "data": [
        {
            "object": "embedding",
            "index": 0,
            "embedding": [
                -0.009171937,
                0.018715322,
                ...
                -0.0016804502
            ]
        }
    ],
    "model": "ada",
    "usage": {
        "prompt_tokens": 7,
        "total_tokens": 7
    }
}
```

In this approach, your application code is responsible for connecting to a model, generating embeddings, and handling the response.

Vector query request

This section shows you the basic structure of a vector query. You can use the Azure portal, REST APIs, or the Azure SDKs to formulate a vector query.

If you're migrating from [2023-07-01-PublicPreview](#), there are breaking changes. For more information, see [Upgrade to the latest REST API](#).

2025-09-01

The stable version supports:

- `vectorQueries` is the construct for vector search.
- `vectorQueries.kind` set to `vector` for a vector array or `text` if the input is a string and if you [have a vectorizer](#).
- `vectorQueries.vector` is the query (a vector representation of text or an image).
- `vectorQueries.exhaustive` (optional) invokes exhaustive KNN at query time, even if the field is indexed for HNSW.
- `vectorQueries.fields` (optional) targets specific fields for query execution (up to 10 per query).
- `vectorQueries.weight` (optional) specifies the relative weight of each vector query included in search operations. For more information, see [Vector weighting](#).
- `vectorQueries.k` is the number of matches to return.

In the following example, the vector is a representation of this string: `"what Azure services support full text search"`. The query targets the `contentVector` field and returns `k` results. The actual vector has 1,536 embeddings, which are trimmed in this example for readability.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
Content-Type: application/json
api-key: {{admin-api-key}}
{
    "count": true,
    "select": "title, content, category",
    "vectorQueries": [
        {
            "kind": "vector",
            "vector": [
                -0.009154141,
                0.018708462,
                ...
            ]
        }
    ]
}
```

```
        -0.02178128,
        -0.00086512347
    ],
    "exhaustive": true,
    "fields": "contentVector",
    "weight": 0.5,
    "k": 5
}
]
}
```

Vector query response

In Azure AI Search, query responses consist of all `retrievable` fields by default. However, it's common to limit search results to a subset of `retrievable` fields by listing them in a `select` statement.

In a vector query, carefully consider whether you need to vector fields in a response. Vector fields aren't human readable, so if you're pushing a response to a web page, you should choose nonvector fields that represent the result. For example, if the query executes against `contentVector`, you could return `content` instead.

If you want vector fields in the result, here's an example of the response structure. `contentVector` is a string array of embeddings, which are trimmed in this example for readability. The search score indicates relevance. Other nonvector fields are included for context.

JSON

```
{
    "@odata.count": 3,
    "value": [
        {
            "@search.score": 0.80025613,
            "title": "Azure Search",
            "category": "AI + Machine Learning",
            "contentVector": [
                -0.0018343845,
                0.017952163,
                0.0025753193,
                ...
            ]
        },
        {
            "@search.score": 0.78856903,
            "title": "Azure Application Insights",
            "category": "Management + Governance",
        }
    ]
}
```

```

    "contentVector": [
        -0.016821077,
        0.0037742127,
        0.016136652,
        ...
    ],
},
{
    "@search.score": 0.78650564,
    "title": "Azure Media Services",
    "category": "Media",
    "contentVector": [
        -0.025449317,
        0.0038463024,
        -0.02488436,
        ...
    ]
}
]
}

```

Key points:

- `k` determines how many nearest neighbor results are returned, in this case, three. Vector queries always return `k` results, assuming at least `k` documents exist, even if some documents have poor similarity. This is because the algorithm finds any `k` nearest neighbors to the query vector.
- The [vector search algorithm](#) determines the `@search.score`.
- Fields in search results are either all `retrievable` fields or fields in a `select` clause. During vector query execution, matching is made on vector data alone. However, a response can include any `retrievable` field in an index. Because there's no facility for decoding a vector field result, the inclusion of nonvector text fields is helpful for their human-readable values.

Multiple vector fields

You can set the `vectorQueries.fields` property to multiple vector fields. The vector query executes against each vector field that you provide in the `fields` list. You can specify up to 10 fields.

When querying multiple vector fields, ensure that each one contains embeddings from the same embedding model. The query should also be generated from the same embedding model.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
Content-Type: application/json
api-key: {{admin-api-key}}
{
  "count": true,
  "select": "title, content, category",
  "vectorQueries": [
    {
      "kind": "vector",
      "vector": [
        -0.009154141,
        0.018708462,
        . . .
        -0.02178128,
        -0.00086512347
      ],
      "exhaustive": true,
      "fields": "contentVector, titleVector",
      "k": 5
    }
  ]
}
```

Multiple vector queries

Multi-query vector search sends multiple queries across multiple vector fields in your search index. This type of query is commonly used with models such as [CLIP](#) for multimodal search, where the same model can vectorize both text and images.

The following query example looks for similarity in both `myImageVector` and `myTextVector` but sends two respective query embeddings, each executing in parallel. The result of this query is scored using [reciprocal rank fusion](#) (RRF).

- `vectorQueries` provides an array of vector queries.
- `vector` contains the image vectors and text vectors in the search index. Each instance is a separate query.
- `fields` specifies which vector field to target.
- `k` is the number of nearest neighbor matches to include in results.

JSON

```
{
  "count": true,
  "select": "title, content, category",
```

```
"vectorQueries": [
  {
    "kind": "vector",
    "vector": [
      -0.009154141,
      0.018708462,
      ...
      -0.02178128,
      -0.00086512347
    ],
    "fields": "myimagevector",
    "k": 5
  },
  {
    "kind": "vector"
    "vector": [
      -0.002222222,
      0.018708462,
      -0.013770515,
      ...
    ],
    "fields": "mytextvector",
    "k": 5
  }
]
```

Search indexes can't store images. Assuming that your index includes a field for the image file, the search results would include a combination of text and images.

Query with integrated vectorization

This section shows a vector query that invokes the [integrated vectorization](#) to convert a text or [image query](#) into a vector. We recommend the stable [2025-09-01](#) REST API, Search Explorer, or newer Azure SDK packages for this feature.

A prerequisite is a search index that has a [vectorizer configured and assigned](#) to a vector field. The vectorizer provides connection information to an embedding model used at query time.

Azure portal

Search Explorer supports integrated vectorization at query time. If your index contains vector fields and has a vectorizer, you can use the built-in text-to-vector conversion.

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. From the left menu, select **Search management > Indexes**, and then select your index.

3. Select the **Vector profiles** tab to confirm that you have a vectorizer.

The screenshot shows the Azure AI Search interface for an index named 'vector-20240531'. The 'Vector profiles' tab is selected. A table lists a single profile:

Profile name	Algorithm	Vectorizer	Compression
vector-20240531-azureOpenAi-text-...	vector-20240531-algorithm	vector-20240531-azureOpenAi-text-...	

4. Select the **Search explorer** tab. Using the default query view, you can enter a text string into the search bar. The built-in vectorizer converts your string into a vector, performs the search, and returns results.

Alternatively, you can select **View > JSON view** to view or modify the query. If vectors are present, Search Explorer sets up a vector query automatically. You can use the JSON view to select fields for use in the search and response, add filters, and construct more advanced queries, such as [hybrid queries](#). To see a JSON example, select the REST API tab in this section.

Number of ranked results in a vector query response

A vector query specifies the `k` parameter, which determines how many matches are returned in the results. The search engine always returns `k` number of matches. If `k` is larger than the number of documents in the index, the number of documents determines the upper limit of what can be returned.

If you're familiar with full-text search, you know to expect zero results if the index doesn't contain a term or phrase. However, in vector search, the search operation identifies nearest neighbors and always return `k` results, even if the nearest neighbors aren't that similar. It's possible to get results for nonsensical or off-topic queries, especially if you aren't using prompts to set boundaries. Less relevant results have a worse similarity score, but they're still the "nearest" vectors if there isn't anything closer. Therefore, a response with no meaningful results can still return `k` results, but each result's similarity score would be low.

A [hybrid approach](#) that includes full-text search can mitigate this problem. Another solution is to set a minimum threshold on the search score, but only if the query is a pure single vector query. Hybrid queries aren't conducive to minimum thresholds because the RRF ranges are much smaller and more volatile.

Query parameters that affect result count include:

- `"k": n` results for vector-only queries.
- `"top": n` results for hybrid queries that include a `search` parameter.

Both `k` and `top` are optional. When unspecified, the default number of results in a response is 50. You can set `top` and `skip` to [page through more results](#) or change the default.

Ranking algorithms used in a vector query

The ranking of results is computed by either:

- The similarity metric.
- RRF if there are multiple sets of search results.

Similarity metric

The similarity metric specified in the index `vectorSearch` section for a vector-only query. Valid values are `cosine`, `euclidean`, and `dotProduct`.

Azure OpenAI embedding models use cosine similarity, so if you're using Azure OpenAI embedding models, `cosine` is the recommended metric. Other supported ranking metrics include `euclidean` and `dotProduct`.

RRF

Multiple sets are created if the query targets multiple vector fields, runs multiple vector queries in parallel, or is a hybrid of vector and full-text search, with or without [semantic ranking](#).

During query execution, a vector query can only target one internal vector index. For [multiple vector fields](#) and [multiple vector queries](#), the search engine generates multiple queries that target the respective vector indexes of each field. The output is a set of ranked results for each query, which are fused using RRF. For more information, see [Relevance scoring using Reciprocal Rank Fusion](#).

Vector weighting

Add a `weight` query parameter to specify the relative weight of each vector query included in search operations. This value is used when combining the results of multiple ranking lists produced by two or more vector queries in the same request, or from the vector portion of a hybrid query.

The default is 1.0, and the value must be a positive number larger than zero.

Weights are used when calculating the [RRF scores](#) of each document. The calculation is a multiplier of the `weight` value against the rank score of the document within its respective result set.

The following example is a hybrid query with two vector query strings and one text string. Weights are assigned to the vector queries. The first query is 0.5 or half the weight, reducing its importance in the request. The second vector query is twice as important.

HTTP

```
POST https://[service-name].search.windows.net/indexes/[index-name]/docs/search?  
api-version=2025-09-01  
  
{  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [1.0, 2.0, 3.0],  
            "fields": "my_first_vector_field",  
            "k": 10,  
            "weight": 0.5  
        },  
        {  
            "kind": "vector",  
            "vector": [4.0, 5.0, 6.0],  
            "fields": "my_second_vector_field",  
            "k": 10,  
            "weight": 2.0  
        }  
    ],  
    "search": "hello world"  
}
```

Vector weighting applies to vectors only. The text query in this example, `"hello world"`, has an implicit neutral weight of 1.0. However, in a hybrid query, you can increase or decrease the importance of text fields by setting [maxTextRecallSize](#).

Set thresholds to exclude low-scoring results (preview)

Because nearest neighbor search always returns the requested `k` neighbors, it's possible to get multiple low-scoring matches as part of meeting the `k` number requirement on search results. To exclude low-scoring search results, you can add a `threshold` query parameter that filters out results based on a minimum score. Filtering occurs before [fusing results](#) from different recall sets.

This parameter is in preview. We recommend the [2024-05-01-preview](#) REST API version.

In this example, all matches that score below 0.8 are excluded from vector search results, even if the number of results falls below `k`.

HTTP

```
POST https://[service-name].search.windows.net/indexes/[index-name]/docs/search?  
api-version=2024-05-01-preview  
Content-Type: application/json  
api-key: [admin key]  
  
{  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [1.0, 2.0, 3.0],  
            "fields": "my-cosine-field",  
            "threshold": {  
                "kind": "vectorSimilarity",  
                "value": 0.8  
            }  
        }  
    ]  
}
```

Next steps

As a next step, review vector query code examples in [Python](#), [C#](#) or [JavaScript](#).

Multi-vector field support in Azure AI Search

08/27/2025

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

The multi-vector field support feature in Azure AI Search enables you to index multiple child vectors within a single document field. This feature is valuable for use cases like multimodal data or long-form documents, where representing the content with a single vector would lead to loss of important detail.

Understanding multi-vector field support

Traditionally, vector types, for example `Collection(Edm.Single)` could only be used in top-level fields. With the introduction of multi-vector field support, you can now use vector types in nested fields of complex collections, effectively allowing multiple vectors to be associated with a single document.

A single document can include up to 100 vectors in total, across all complex collection fields. Vector fields can only be nested one level deep.

Index definition with multi-vector field

No new index properties are needed for this feature. Here's a sample index definition:

JSON

```
{  
  "name": "multivector-index",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true,  
      "searchable": true  
    },
```

```
{  
    "name": "title",  
    "type": "Edm.String",  
    "searchable": true  
},  
{  
    "name": "description",  
    "type": "Edm.String",  
    "searchable": true  
},  
{  
    "name": "descriptionEmbedding",  
    "type": "Collection(Edm.Single)",  
    "dimensions": 3,  
    "searchable": true,  
    "retrievable": true,  
    "vectorSearchProfile": "hnsw"  
},  
{  
    "name": "scenes",  
    "type": "Collection(Edm.ComplexType)",  
    "fields": [  
        {  
            "name": "embedding",  
            "type": "Collection(Edm.Single)",  
            "dimensions": 3,  
            "searchable": true,  
            "retrievable": true,  
            "vectorSearchProfile": "hnsw"  
        },  
        {  
            "name": "timestamp",  
            "type": "Edm.Int32",  
            "retrievable": true  
        },  
        {  
            "name": "description",  
            "type": "Edm.String",  
            "searchable": true,  
            "retrievable": true  
        },  
        {  
            "name": "framePath",  
            "type": "Edm.String",  
            "retrievable": true  
        }  
    ]  
}  
]
```

Sample ingest document

Here's a sample document that illustrates how you might use multi-vector fields in practice:

JSON

```
{  
  "id": "123",  
  "title": "Non-Existent Movie",  
  "description": "A fictional movie for demonstration purposes.",  
  "descriptionEmbedding": [1, 2, 3],  
  "releaseDate": "2025-08-01",  
  "scenes": [  
    {  
      "embedding": [4, 5, 6],  
      "timestamp": 120,  
      "description": "A character is introduced.",  
      "framePath": "nonexistentmovie\\scenes\\scene120.png"  
    },  
    {  
      "embedding": [7, 8, 9],  
      "timestamp": 2400,  
      "description": "The climax of the movie.",  
      "framePath": "nonexistentmovie\\scenes\\scene2400.png"  
    }  
  ]  
}
```

In this example, the scenes field is a complex collection containing multiple vectors (the embedding fields), along with other associated data. Each vector represents a scene from the movie and could be used to find similar scenes in other movies, among other potential use cases.

Querying with multi-vector field support

The multi-vector field support feature introduces some changes to the query mechanism in Azure AI Search. However, the main querying process remains largely the same. Previously, `vectorQueries` could only target vector fields defined as top-level index fields. With this feature, we're relaxing this restriction and allowing `vectorQueries` to target fields that are nested within a collection of complex types (up to one level deep). Additionally, a new query time parameter is available: `perDocumentVectorLimit`.

- Setting `perDocumentVectorLimit` to `1` ensures that at most one vector per document is matched, guaranteeing that results come from distinct documents.
- Setting `perDocumentVectorLimit` to `0` (unlimited) allows multiple relevant vectors from the same document to be matched.

JSON

```
{  
  "vectorQueries": [  
    {  
      "kind": "text",  
      "text": "whales swimming",  
      "K": 50,  
      "fields": "scenes/embedding",  
      "perDocumentVectorLimit": 0  
    }  
  ],  
  "select": "title, scenes/timestamp, scenes/framePath"  
}
```

Ranking across multiple vectors in a single field

When multiple vectors are associated with a single document, Azure AI Search uses the maximum score among them for ranking. The system uses the most relevant vector to score each document, which prevents dilution by less relevant ones.

Retrieving the relevant elements in a collection

When a collection of complex types is included in the `$select` parameter, only the elements that matched the vector query are returned. This is useful for retrieving associated metadata such as timestamps, text descriptions, or image paths.

 **Note**

To reduce payload size, avoid including the vector values themselves in the `$select` parameter. Consider omitting vector storage entirely if unnecessary.

Limitations

- Semantic ranker isn't supported for nested chunks within a complex field. Therefore, the semantic ranker doesn't support nested vectors in multi-vector fields.

Debugging multi-vector queries (Preview)

When a document includes multiple embedded vectors, such as text and image embeddings in different subfields, the system uses the highest vector score across all elements to rank the document.

To debug how each vector contributed, use the `innerHits` debug mode (available in the latest preview REST API).

JSON

```
POST /indexes/my-index/docs/search?api-version=2025-08-01-preview
{
  "vectorQueries": [
    {
      "kind": "vector",
      "field": "keyframes.imageEmbedding",
      "kNearestNeighborsCount": 5,
      "vector": [ /* query vector */ ]
    }
  ],
  "debug": "innerHits"
}
```

Example response shape

JSON

```
"@search.documentDebugInfo": {
  "innerHits": {
    "keyframes": [
      {
        "ordinal": 0,
        "vectors": [
          {
            "imageEmbedding": {
              "searchScore": 0.958,
              "vectorSimilarity": 0.956
            },
            "textEmbedding": {
              "searchScore": 0.958,
              "vectorSimilarity": 0.956
            }
          }
        ]
      },
      {
        "ordinal": 1,
        "vectors": [
          {
            "imageEmbedding": null,
            "textEmbedding": {
              "searchScore": 0.872,
              "vectorSimilarity": 0.869
            }
          }
        ]
      }
    ]
  }
}
```

```
        }
    ]
}
```

Field descriptions

[Expand table](#)

Field	Description
<code>ordinal</code>	Zero-based index of the element inside the collection.
<code>vectors</code>	One entry per searchable vector field contained in the element.
<code>searchScore</code>	Final score for that field, after any rescaling and boosts.
<code>vectorSimilarity</code>	Raw similarity returned by the distance function.

 Note

`innerHits` currently reports only vector fields.

Relationship to debug=vector

Here are some facts about this property:

- The existing `debug=vector` switch remains unchanged.
- When used with multi-vector fields, `@search.document DebugInfo.vector.subscore` shows the maximum score used to rank the parent document, but not per-element detail.
- Use `innerHits` to gain insight into how individual elements contributed to the score.

Configure a vectorizer in a search index

In Azure AI Search, a *vectorizer* is a component that performs vectorization using a deployed embedding model on Azure OpenAI or Azure Vision in Foundry Tools. It converts text (or images) to vectors during query execution.

It's defined in a [search index](#), it applies to searchable vector fields, and it's used at query time to generate an embedding for a text or image query input. If instead you need to vectorize content as part of the indexing process, refer to [integrated vectorization](#). For built-in vectorization during indexing, you can configure an indexer and skillset that calls an embedding model for your raw text or image content.

To add a vectorizer to search index, you can use the index designer in Azure portal, or call the [Create or Update Index REST API](#), or use any Azure SDK package that's updated to provide this feature.

Vectorizers are now generally available as long as you use a generally available skill-vectorizer pair. [AzureOpenAIEmbedding vectorizer](#) and [AzureOpenAIEmbedding skill](#) are generally available. The custom [Web API vectorizer](#) is also generally available.

[Azure Vision vectorizer](#), [Microsoft Foundry model catalog vectorizer](#), and their equivalent skills are still in preview. Your skillset must specify [2024-05-01-preview REST API](#) to use preview skills and vectorizers.

Prerequisites

- An [index with searchable vector fields](#) on Azure AI Search.
- A deployed embedding model (see the next section).
- Permissions to use the embedding model. On Azure OpenAI, the caller must have [Cognitive Services OpenAI User](#) permissions. Or, you can provide an API key.
- [Visual Studio Code](#) with a [REST client](#) to send the query and accept a response.

We recommend that you [enable diagnostic logging](#) on your search service to confirm vector query execution.

Supported embedding models

The following table lists the embedding models that can be used with a vectorizer. Because you must use the [same embedding model for indexing and queries](#), vectorizers are paired with

skills that generate embeddings during indexing. The table lists the skill associated with a particular vectorizer.

[] [Expand table](#)

Vectorizer kind	Model names	Model provider	Associated skill
azureOpenAI	text-embedding-ada-002	Azure OpenAI	AzureOpenAIEmbedding skill
	text-embedding-3		
aml	Cohere-embed-v3 Cohere-embed-v4 ¹	Foundry model catalog	AML skill
aiServicesVision	Multimodal embeddings 4.0 API	Azure Vision (through a Foundry resource)	Azure Vision multimodal embeddings skill
customWebApi	Any embedding model	Hosted externally	Custom Web API skill

¹ At this time, you can only specify `embed-v-4-0` programmatically through the [AML skill](#) or [Microsoft Foundry model catalog vectorizer](#), not through the Azure portal. However, you can use the portal to manage the skillset or vectorizer afterward.

Try a vectorizer with sample data

The [Import data \(new\) wizard](#) reads files from Azure Blob storage, creates an index with chunked and vectorized fields, and adds a vectorizer. By design, the vectorizer that's created by the wizard is set to the same embedding model used to index the blob content.

1. Upload sample data files to a container on Azure Storage. We used some [small text files from NASA's earth book](#) to test these instructions on a free search service.
2. Run the [Import data \(new\) wizard](#), choosing the blob container for the data source.

Configure your Azure Blob Storage

Connect to Azure Blob Storage to access your structured and unstructured data files, including PDFs. [Learn more](#)

Subscription *	my-subscription
Storage account *	mystorageaccount
Blob container * ⓘ	nasa-ebooks-text
Blob folder ⓘ	your/folder/here
Parsing mode	Default

Enable document layout detection (Preview) ⓘ

Enable deletion tracking. ⓘ

Authenticate using managed identity. [Learn more](#)

[Previous](#) [Next](#) 

3. Choose an existing deployment of **text-embedding-ada-002**. This model generates embeddings during indexing and is also used to configure the vectorizer used during queries.

Vectorize your text

Connect to an Azure OpenAI, AI Foundry or an Azure AI service and select an embedding model or multi-service account for vector generation. [Learn more](#)

Kind	Azure OpenAI
Subscription *	my-subscription
Azure OpenAI service * ⓘ	my-aoai-resource
	Create a new Azure OpenAI service
Model deployment * ⓘ	text-embedding-ada-002

Authentication type ⓘ API key System assigned identity User assigned identity

I acknowledge that connecting to an Azure OpenAI service will incur additional costs to my account. [View pricing](#)

[Previous](#) [Next](#) 

4. After the wizard is finished and all indexer processing is complete, you should have an index with a searchable vector field. The field's JSON definition looks like this:

JSON

```
{  
  "name": "vector",  
  "type": "Collection(Edm.Single)",  
  "searchable": true,
```

```
        "retrievable": true,  
        "dimensions": 1536,  
        "vectorSearchProfile": "vector-nasa-ebook-text-profile"  
    }  
}
```

5. You should also have a vector profile and a vectorizer, similar to the following example:

JSON

```
"profiles": [  
    {  
        "name": "vector-nasa-ebook-text-profile",  
        "algorithm": "vector-nasa-ebook-text-algorithm",  
        "vectorizer": "vector-nasa-ebook-text-vectorizer"  
    }  
,  
    "vectorizers": [  
        {  
            "name": "vector-nasa-ebook-text-vectorizer",  
            "kind": "azureOpenAI",  
            "azureOpenAIParameters": {  
                "resourceUri": "https://my-fake-azure-openai-resource.openai.azure.com",  
                "deploymentId": "text-embedding-ada-002",  
                "modelName": "text-embedding-ada-002",  
                "apiKey": "00000000000000000000000000000000",  
                "authIdentity": null  
            },  
            "customWebApiParameters": null  
        }  
    ]  
]
```

6. Skip ahead to [test your vectorizer](#) for text-to-vector conversion during query execution.

Define a vectorizer and vector profile

This section explains the modifications to an index schema for defining a vectorizer manually.

1. Use [Create or Update Index](#) to add `vectorizers` to a search index.
2. Add the following JSON to your index definition. The vectorizers section provides connection information to a deployed embedding model. This step shows two vectorizer examples so that you can compare an Azure OpenAI embedding model and a custom web API side by side.

JSON

```
"vectorizers": [  
    {  
        "name": "my_azure_open_ai_vectorizer",  
        "kind": "azureOpenAI",  
        "azureOpenAIParameters": {  
            "resourceUri": "https://my-fake-azure-openai-resource.openai.azure.com",  
            "deploymentId": "text-embedding-ada-002",  
            "modelName": "text-embedding-ada-002",  
            "apiKey": "00000000000000000000000000000000",  
            "authIdentity": null  
        },  
        "customWebApiParameters": null  
    }  
]
```

```

    "kind": "azureOpenAI",
    "azureOpenAIParameters": {
        "resourceUri": "https://url.openai.azure.com",
        "deploymentId": "text-embedding-ada-002",
        "modelName": "text-embedding-ada-002",
        "apiKey": "mytopsecretkey"
    }
},
{
    "name": "my_custom_vectorizer",
    "kind": "customWebApi",
    "customVectorizerParameters": {
        "uri": "https://my-endpoint",
        "authResourceId": " ",
        "authIdentity": " "
    }
}
]

```

3. In the same index, add a vector profiles section that specifies one of your vectorizers.

Vector profiles also require a [vector search algorithm](#) used to create navigation structures.

JSON

```

"profiles": [
    {
        "name": "my_vector_profile",
        "algorithm": "my_hnsw_algorithm",
        "vectorizer": "my_azure_open_ai_vectorizer"
    }
]

```

4. Assign a vector profile to a vector field. The following example shows a fields collection with the required key field, a title string field, and two vector fields with a vector profile assignment.

JSON

```

"fields": [
    {
        "name": "ID",
        "type": "Edm.String",
        "key": true,
        "sortable": true,
        "analyzer": "keyword"
    },
    {
        "name": "title",
        "type": "Edm.String"
    },
    {

```

```
        "name": "vector",
        "type": "Collection(Edm.Single)",
        "dimensions": 1536,
        "vectorSearchProfile": "my_vector_profile",
        "searchable": true,
        "retrievable": true
    },
    {
        "name": "my-second-vector",
        "type": "Collection(Edm.Single)",
        "dimensions": 1024,
        "vectorSearchProfile": "my_vector_profile",
        "searchable": true,
        "retrievable": true
    }
]
```

Test a vectorizer

Use a search client to send a query through a vectorizer. This example assumes Visual Studio Code with a REST client and a [sample index](#).

1. In Visual Studio Code, provide a search endpoint and [search query API key](#):

```
HTTP
```

```
@baseUrl:
@queryApiKey: 00000000000000000000000000000000
```

2. Paste in a [vector query request](#).

```
HTTP
```

```
### Run a query
POST {{baseUrl}}/indexes/vector-nasa-ebook-txt/docs/search?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{queryApiKey}}


{
    "count": true,
    "select": "title,chunk",
    "vectorQueries": [
        {
            "kind": "text",
            "text": "what cloud formations exists in the troposphere",
            "fields": "vector",
            "k": 3,
            "exhaustive": true
        }
    ]
}
```

```
        ]  
    }
```

Key points about the query include:

- `"kind": "text"` tells the search engine that the input is a text string, and to use the vectorizer associated with the search field.
- `"text": "what cloud formations exists in the troposphere"` is the text string to vectorize.
- `"fields": "vector"` is the name of the field to query over. If you use the sample index produced by the wizard, the generated vector field is named `vector`.

3. Send the request. You should get three `k` results, where the first result is the most relevant.

Notice that there are no vectorizer properties to set at query time. The query reads the vectorizer properties, as per the vector profile field assignment in the index.

Check logs

If you enabled diagnostic logging for your search service, run a Kusto query to confirm query execution on your vector field:

```
Kusto
```

```
OperationEvent  
| where TIMESTAMP > ago(30m)  
| where Name == "Query.Search" and AdditionalInfo["QueryMetadata"]["Vectors"] has  
"TextLength"
```

Best practices

If you're setting up an Azure OpenAI vectorizer, consider the same [best practices](#) that we recommend for the Azure OpenAI embedding skill.

See also

- [Integrated vectorization \(preview\)](#)

Add a filter to a vector query in Azure AI Search

09/16/2025

! Note

`strictPostFilter` is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

`prefilter` and `postfilter` are generally available in the [latest stable REST API version](#).

In Azure AI Search, you can use a [filter expression](#) to add inclusion or exclusion criteria to a [vector query](#). You can also specify a filtering mode that applies the filter:

- Before query execution, known as *prefiltering*.
- After query execution, known as *postfiltering*.
- After the global top-`k` results are identified, known as *strict postfiltering* (preview).

This article uses REST for illustration. For code samples in other languages and end-to-end solutions that include vector queries, see the [azure-search-vector-samples](#) GitHub repository.

You can also use [Search Explorer](#) in the Azure portal to query vector content. In the JSON view, you can add filters and specify the filter mode.

How filtering works in vector queries

Azure AI Search uses the Hierarchical Navigable Small World (HNSW) algorithm for Approximate Nearest Neighbor (ANN) search, storing HNSW graphs across multiple shards. Each shard contains a portion of the entire index.

Filters apply to `filterable nonvector` fields, either string or numeric, to include or exclude search documents based on filter criteria. Vector fields themselves aren't filterable, but you can use filters on other fields in the same index to narrow the documents considered for vector search. If your index lacks suitable text or numeric fields, check for document metadata that might help with filtering, such as `LastModified` or `CreatedBy` properties.

The `vectorFilterMode` parameter controls where filter operations are applied during the stages of search, which affects how the results are filtered to a subset of items (such as by category, tag, or other attributes) and impacts latency, recall, and throughput. There are three modes:

- `preFilter` applies the filter *during* HNSW traversal on each shard. This mode maximizes recall but can traverse more of the graph, increasing CPU and latency for highly selective filters.
- `postFilter` runs HNSW traversal and filtering on each shard independently, intersects results at the shard level, and then aggregates the top `k` from each shard into a global top `k`. This mode can create false negatives for highly selective filters or small `k` values.
- `strictPostFilter` (preview) finds the unfiltered global top `k` *before* applying the filter. This mode has the highest risk of returning false negatives for highly selective filters and small `k` values.

For more information about these modes, see [Set the filter mode](#).

Define a filter

Filters determine the scope of vector queries and are defined using [Documents - Search Post \(REST API\)](#). Unless you want to use a preview feature, use the latest stable version of the [Search Service REST APIs](#) to formulate the request.

This REST API provides:

- `filter` for the criteria.
- `vectorFilterMode` to specify when the filter is applied during the vector query. For supported modes, see [Set the filter mode](#).

HTTP

```
POST https://{{search-endpoint}}/indexes/{{index-name}}/docs/search?api-version={{api-version}}
Content-Type: application/json
api-key: {{admin-api-key}}


{  
    "count": true,  
    "select": "title, content, category",  
    "filter": "category eq 'Databases'",  
    "vectorFilterMode": "preFilter",  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [  
                {{vector}}  
            ]  
        }  
    ]  
}
```

```
-0.009154141,  
0.018708462,  
. . . // Trimmed for readability  
-0.02178128,  
-0.00086512347  
],  
"fields": "contentVector",  
"k": 50  
}  
]  
}
```

In this example, the vector embedding targets the `contentVector` field, and the filter criteria apply to `category`, a filterable text field. Because the `preFilter` mode is used, the filter is applied before the search engine runs the query, so only documents in the `Databases` category are considered during the vector search.

Set the filter mode

The `vectorFilterMode` parameter determines when and how the filter is applied relative to vector query execution. You can use the following modes:

- `preFilter` (recommended)
- `postFilter`
- `strictPostFilter` (preview)

(!) Note

`preFilter` is the default for indexes created after approximately October 15, 2023. For indexes created before this date, `postFilter` is the default. To use `preFilter` and other advanced vector features, such as vector compression, you must recreate your index.

You can test compatibility by sending a vector query with `"vectorFilterMode": "preFilter"` on the `2023-10-01-preview` REST API version or later. If the query fails, your index doesn't support `preFilter`.

preFilter

Prefiltering applies filters before query execution, which reduces the candidate set for the vector search algorithm. The top-`k` results are then selected from this filtered set.

In a vector query, `prefilter` is the default mode because it favors recall and quality over latency.

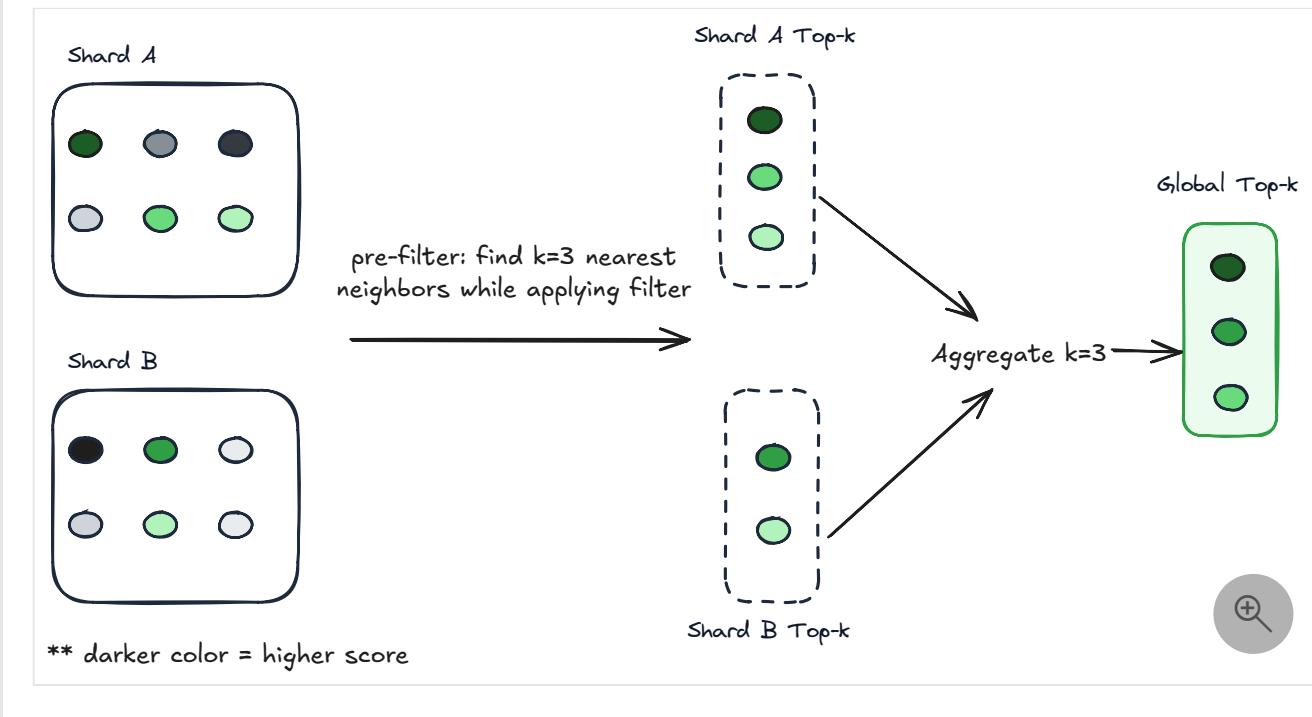
How this mode works

1. On each shard, apply the filter predicate *during* HNSW traversal, expanding the graph until `k` candidates are found.
2. Produce the prefiltered local top-`k` results per shard.
3. Aggregate the filtered results into a global top-`k` result set.

Effect of this mode

Traversal expands the search surface to find more filtered candidates, especially if the filter is selective. This produces the most similar top-`k` results across all shards. Each shard identifies the `k` results that satisfy the filter predicate.

Prefiltering guarantees that `k` results are returned if they exist in the index. For highly selective filters, this can cause a significant portion of the graph to be traversed, increasing computation cost and latency while reducing throughput. If your filter is highly selective (has very few matches), consider using `exhaustive: true` to perform exhaustive search.



Comparison table

Mode	Recall (filtered results)	Computational cost	Risk of false negatives	When to use
preFilter	Very high	Higher (increases with filter selectivity and complexity)	No risk	Recommended default for all scenarios, especially when recall is critical (sensitive search domains), when using selective filters, or when using small k .
postFilter	Medium to high (decreases with filter selectivity)	Similar to unfiltered but increases with filter complexity	Moderate (can miss matches per shard)	An option for filters that aren't too selective and for higher- k queries.
strictPostFilter	Lowest (decreases most quickly with filter selectivity)	Similar to unfiltered	Highest (can return zero results for selective filters or small k)	An option for faceted search applications where surfacing more results after filter application impacts the user experience more than the risk of false negatives. Don't use with small k .

Benchmark testing of prefiltering and postfiltering

ⓘ Important

This section applies to prefiltering and postfiltering, not strict postfiltering.

To understand the conditions under which one filter mode performs better than the other, we ran a series of tests to evaluate query outcomes over small, medium, and large indexes.

- Small (100,000 documents, 2.5-GB index, 1,536 dimensions)
- Medium (1 million documents, 25-GB index, 1,536 dimensions)
- Large (1 billion documents, 1.9-TB index, 96 dimensions)

For the small and medium workloads, we used a Standard 2 (S2) service with one partition and one replica. For the large workload, we used a Standard 3 (S3) service with 12 partitions and one replica.

Indexes had an identical construction: one key field, one vector field, one text field, and one numeric filterable field. The following index is defined using the 2023-11-03 syntax.

Python

```
def get_index_schema(self, index_name, dimensions):
    return {
        "name": index_name,
        "fields": [
            {"name": "id", "type": "Edm.String", "key": True, "searchable": True},
            {"name": "content_vector", "type": "Collection(Edm.Single)"}
        ],
        "dimensions": dimensions,
        "searchable": True, "retrievable": True, "filterable": False,
        "facetable": False, "sortable": False,
        "vectorSearchProfile": "defaulthnsw"},
        {"name": "text", "type": "Edm.String", "searchable": True,
        "filterable": False, "retrievable": True,
        "sortable": False, "facetable": False},
        {"name": "score", "type": "Edm.Double", "searchable": False,
        "filterable": True,
        "retrievable": True, "sortable": True, "facetable": True}
    ],
    "vectorSearch": {
        "algorithms": [
            {
                "name": "defaulthnsw",
                "kind": "hnsw",
                "hnswParameters": { "metric": "euclidean" }
            }
        ],
        "profiles": [
            {
                "name": "defaulthnsw",
                "algorithm": "defaulthnsw"
            }
        ]
    }
}
```

In queries, we used an identical filter for both prefilter and postfilter operations. We used a simple filter to ensure that variations in performance were due to filtering mode, not filter complexity.

Outcomes were measured in queries per second (QPS).

Takeaways

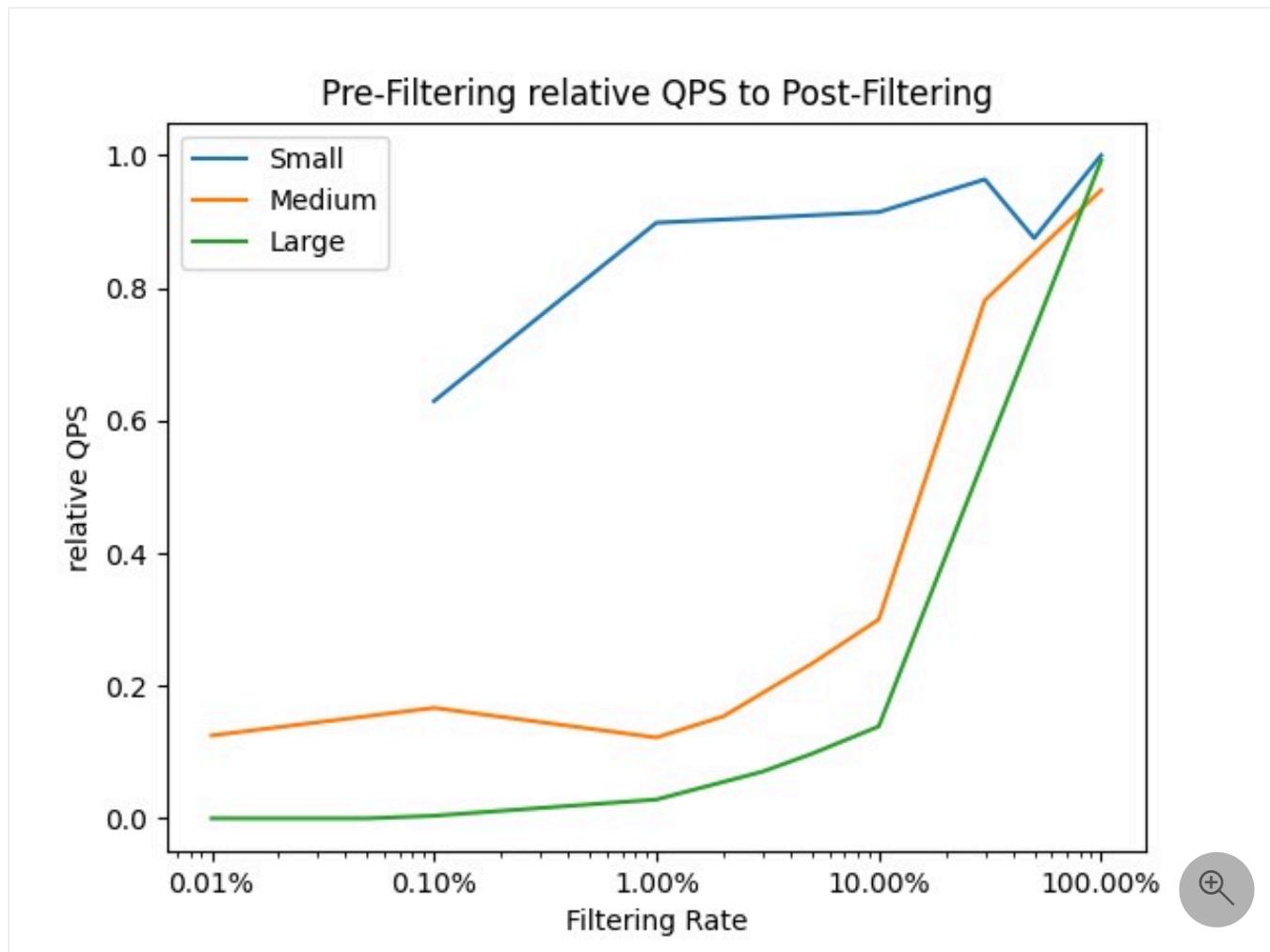
- Prefiltering is almost always slower than postfiltering, except on small indexes where performance is approximately equal.

- On larger datasets, prefiltering is orders of magnitude slower.
- Why is prefilter the default if it's almost always slower? Prefiltering guarantees that k results are returned if they exist in the index, where the bias favors recall and precision over speed.
- Use postfiltering if you:
 - Value speed over selection (postfiltering can return fewer than k results).
 - Use filters that aren't overly selective.
 - Have indexes of sufficient size such that prefiltering performance is unacceptable.

Details

- Given a dataset with 100,000 vectors at 1,536 dimensions:
 - When filtering more than 30% of the dataset, prefiltering and postfiltering were comparable.
 - When filtering less than 0.1% of the dataset, prefiltering was about 50% slower than postfiltering.
- Given a dataset with 1 million vectors at 1,536 dimensions:
 - When filtering more than 30% of the dataset, prefiltering was about 30% slower.
 - When filtering less than 2% of the dataset, prefiltering was about seven times slower.
- Given a dataset with 1 billion vectors at 96 dimensions:
 - When filtering more than 5% of the dataset, prefiltering was about 50% slower.
 - When filtering less than 10% of the dataset, prefiltering was about seven times slower.

The following graph shows prefilter relative QPS, computed as prefilter QPS divided by postfilter QPS.



The vertical axis represents the relative performance of prefiltering compared to postfiltering, expressed as a ratio of QPS (queries per second). For example:

- A value of `0.0` means prefiltering is 100% slower than postfiltering.
- A value of `0.5` means prefiltering is 50% slower.
- A value of `1.0` means prefiltering and post filtering are equivalent.

The horizontal axis represents the filtering rate, or the percentage of candidate documents after applying the filter. For example, a rate of `1.00%` means the filter criteria selected one percent of the search corpus.

Related content

- [Vector search in Azure AI Search](#)
- [Create a vector index](#)
- [Create a vector query](#)

Create a full text query in Azure AI Search

10/03/2025

If you're building a query for [full text search](#), this article provides steps for setting up the request. It also introduces a query structure, and explains how field attributes and linguistic analyzers can affect query outcomes.

Prerequisites

- A [search index](#) with string fields attributed as *searchable*. You can also use an [index alias](#) as the endpoint of your query request.
- Read permissions on the search index. For read access, include a [query API key](#) on the request, or give the caller [Search Index Data Reader](#) permissions.

Example of a full text query request

In Azure AI Search, a query is a read-only request against the docs collection of a single search index, with parameters that both inform query execution and shape the response coming back.

A full text query is specified in a `search` parameter and consists of terms, quote-enclosed phrases, and operators. Other parameters add more definition to the request.

The following [Search POST REST API](#) call illustrates a query request using `search` and other parameters.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-09-01
{
    "search": "NY +view",
    "queryType": "simple",
    "searchMode": "all",
    "searchFields": "HotelName, Description, Address/City, Address/StateProvince,
Tags",
    "select": "HotelName, Description, Address/City, Address/StateProvince, Tags",
    "top": 10,
    "count": true
}
```

Key points

- `search` provides the match criteria, usually whole terms or phrases, with or without operators. Any field that is attributed as *searchable* in the index schema is within scope for a search operation.
- `queryType` sets the parser: *simple*, *full*. The [default simple query parser](#) is optimal for full text search. The [full Lucene query parser](#) is for advanced query constructs like regular expressions, proximity search, fuzzy and wildcard search. This parameter can also be set to *semantic* for [semantic ranking](#) for advanced semantic modeling on the query response.
- `searchMode` specifies whether matches are based on *all* criteria (favors precision) or *any* criteria (favors recall) in the expression. The default is *any*. If you anticipate heavy use of Boolean operators, which is more likely in indexes that contain large text blocks (a content field or long descriptions), be sure to test queries with the `searchMode=Any|All` parameter to evaluate the impact of that setting on Boolean search.
- `searchFields` constrains query execution to specific searchable fields. During development, it's helpful to use the same field list for select and search. Otherwise a match might be based on field values that you can't see in the results, creating uncertainty as to why the document was returned.

Parameters used to shape the response:

- `select` specifies which fields to return in the response. Only fields marked as *retrievable* in the index can be used in a select statement.
- `top` returns the specified number of best-matching documents. In this example, only 10 hits are returned. You can use top and skip (not shown) to page the results.
- `count` tells you how many documents in the entire index match overall, which can be more than what are returned. Valid values are "true" or "false". Defaults to "false". Count is accurate if the index is stable, but will under or over-report any documents that are actively added, updated, or deleted. If you'd like to get only the count without any documents, you can use \$top=0.
- `orderby` is used if you want to sort results by a value, such as a rating or location. Otherwise, the default is to use the relevance score to rank results. A field must be attributed as *sortable* to be a candidate for this parameter.

Choose a client

For early development and proof-of-concept testing, start with the Azure portal or a REST client or a Jupyter notebook. These approaches are interactive, useful for targeted testing, and

help you assess the effects of different properties without having to write any code.

To call search from within an app, use the `Azure.Document.Search` client libraries in the Azure SDKs for .NET, Java, JavaScript, and Python.

Azure portal

In the Azure portal, when you open an index, you can work with Search Explorer alongside the index JSON definition in side-by-side tabs for easy access to field attributes. Check the **Fields** table to see which ones are searchable, sortable, filterable, and facetable while testing queries.

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. In your service, select **Indexes** and choose an index.
3. An index opens to the **Search explorer** tab so that you can query right away. Switch to **JSON view** to specify query syntax.

Here's a full text search query expression that works for the Hotels sample index:

```
JSON

{
    "search": "pool spa +airport",
    "queryType": "simple",
    "searchMode": "any",
    "searchFields": "Description, Tags",
    "select": "HotelName, Description, Tags",
    "top": 10,
    "count": true
}
```

The following screenshot illustrates the query and response:

The screenshot shows the Microsoft Azure AI Search interface for a project named "hotels-sample-index". At the top, there are navigation links for "Home", "Search explorer", "Fields", "CORS", "Scoring profiles", "Semantic configurations", and "Vector profiles". Below this, the "Search explorer" tab is selected. A JSON query editor window is open, displaying the following code:

```
2 |   "search": "pool spa +airport",
3 |   "queryType": "simple",
4 |   "searchMode": "any",
5 |   "searchFields": "Description, Tags",
6 |   "select": "HotelName, Description, Tags",
7 |   "top": 10,
8 |   "count": true
```

To the right of the query editor, a dropdown menu titled "View" is open, showing options: "Query view" (selected), "Image view", and "JSON view". The "Results" section below the query editor displays the search results in JSON format, showing four hotel entries. On the far right, there is a small preview pane showing thumbnail images of the hotel results.

Choose a query type: simple | full

If your query is full text search, a query parser is used to process any text that's passed as search terms and phrases. Azure AI Search offers two query parsers.

- The simple parser understands the [simple query syntax](#). This parser was selected as the default for its speed and effectiveness in free form text queries. The syntax supports common search operators (AND, OR, NOT) for term and phrase searches, and prefix (*) search (as in `sea*` for Seattle and Seaside). A general recommendation is to try the simple parser first, and then move on to full parser if application requirements call for more powerful queries.
- The [full Lucene query syntax](#), enabled when you add `queryType=full` to the request, is based on the [Apache Lucene Parser](#).

Full syntax and simple syntax overlap to the extent that both support the same prefix and Boolean operations, but the full syntax provides more operators. In full, there are more

operators for Boolean expressions, and more operators for advanced queries such as fuzzy search, wildcard search, proximity search, and regular expressions.

Choose query methods

Search is fundamentally a user-driven exercise, where terms or phrases are collected from a search box, or from click events on a page. The following table summarizes the mechanisms by which you can collect user input, along with the expected search experience.

 Expand table

Input	Experience
Search method	A user types the terms or phrases into a search box, with or without operators, and selects Search to send the request. Search can be used with filters on the same request, but not with autocomplete or suggestions.
Autocomplete method	A user types a few characters, and queries are initiated after each new character is typed. The response is a completed string from the index. If the string provided is valid, the user selects Search to send that query to the service.
Suggestions method	As with autocomplete, a user types a few characters and incremental queries are generated. The response is a dropdown list of matching documents, typically represented by a few unique or descriptive fields. If any of the selections are valid, the user selects one and the matching document is returned.
Faceted navigation	A page shows clickable navigation links or breadcrumbs that narrow the scope of the search. A faceted navigation structure is composed dynamically based on an initial query. For example, <code>search=*</code> to populate a faceted navigation tree composed of every possible category. A faceted navigation structure is created from a query response, but it's also a mechanism for expressing the next query. In REST API reference, <code>facets</code> is documented as a query parameter of a Search Documents operation, but it can be used without the <code>search</code> parameter.
Filter method	Filters are used with facets to narrow results. You can also implement a filter behind the page, for example to initialize the page with language-specific fields. In REST API reference, <code>\$filter</code> is documented as a query parameter of a Search Documents operation, but it can be used without the <code>search</code> parameter.

Effect of field attributes on queries

If you're familiar with [query types and composition](#), you might remember that the parameters on a query request depend on field attributes in an index. For example, only fields marked as *searchable* and *retrievable* can be used in queries and search results. When setting the `search`,

`filter`, and `orderby` parameters in your request, you should check attributes to avoid unexpected results.

In the following screenshot of the [hotels sample index](#), only the last two fields **LastRenovationDate** and **Rating** are *sortable*, a requirement for use in an `"$orderby"` only clause.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACTETABLE	SEARCHABLE
霆 HotelId	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Category	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tags	Collection(E...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ParkingIncluded	Edm.Boolean	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
LastRenovationDate	Edm.DateTi...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>		<input type="checkbox"/>
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>

For field attribute definitions, see [Create Index \(REST API\)](#).

Effect of tokens on queries

During indexing, the search engine uses a text analyzer on strings to maximize the potential for finding a match at query time. At a minimum, strings are lower-cased, but depending on the analyzer, might also undergo lemmatization and stop word removal. Larger strings or compound words are typically broken up by whitespace, hyphens, or dashes, and indexed as separate tokens.

The key point is that what you think your index contains, and what's actually in it, can be different. If queries don't return expected results, you can inspect the tokens created by the analyzer through the [Analyze Text \(REST API\)](#). For more information about tokenization and the effect on queries, see [Partial term search and patterns with special characters](#).

Related content

Now that you have a better understanding of how query requests work, try the following quickstarts for hands-on experience.

- [Search explorer](#)
- [Quickstart: Full-text search](#)

Add autocomplete and search suggestions in client apps

Article • 04/14/2025

Search-as-you-type is a common technique for improving query productivity. In Azure AI Search, this experience is supported through *autocomplete*, which finishes a term or phrase based on partial input (for example, completing *micro* with *microchip*, *microscope*, *microsoft*, and any other micro matches). A second user experience is *suggestions*, which produces a short list of matching documents (for example, returning book titles with an ID so that you can link to a detail page about that book). Both autocomplete and suggestions are predicated on a match in the index. The service doesn't offer autocompleted queries or suggestions that return zero results.

To implement these experiences in Azure AI Search:

- Add a `suggester` to an index schema.
- Build a query that calls the [Autocomplete API](#) or [Suggestions API](#) on the request.
- Add a UI control to handle search-as-you-type interactions in your client app. We recommend using an existing JavaScript library for this purpose.

In Azure AI Search, autocompleted queries and suggested results are retrieved from the search index, from selected fields that you register with a suggester. A suggester is part of the index, and it specifies which fields provide content that either completes a query, suggests a result, or does both. When the index is created and loaded, a suggester data structure is created internally to store prefixes used for matching on partial queries. For suggestions, choosing suitable fields that are unique, or at least not repetitive, is essential to the experience. For more information, see [Configure a suggester](#).

The remainder of this article is focused on queries and client code. It uses JavaScript and C# to illustrate key points. REST API examples are used to concisely present each operation. For end-to-end code samples, see [Add search to a web site with .NET](#).

Set up a request

Elements of a request include one of the search-as-you-type APIs, a partial query, and a suggester. The following script illustrates components of a request, using the Autocomplete REST API as an example.

HTTP

```
POST /indexes/myboxgames/docs/autocomplete?search&api-version=2024-07-01
{
```

```
"search": "minecraf",
"suggesterName": "sg"
}
```

The `suggesterName` parameter gives you the suggester-aware fields used to complete terms or suggestions. For suggestions in particular, the field list should be composed of suggestions that offer clear choices among matching results. On a site that sells computer games, the field might be the game title.

The `search` parameter provides the partial query, where characters are fed to the query request through the jQuery Autocomplete control. In the previous example, *minecraf* is a static illustration of what the control might pass in.

The APIs don't impose minimum length requirements on the partial query; it can be as little as one character. However, jQuery Autocomplete provides a minimum length. A minimum of two or three characters is typical.

Matches are on the beginning of a term anywhere in the input string. Given *the quick brown fox*, both autocomplete and suggestions match on partial versions of *the*, *quick*, *brown*, or *fox* but not on partial infix terms like *rown* or *ox*. Furthermore, each match sets the scope for downstream expansions. A partial query of *quick br* matches on *quick brown* or *quick bread*, but neither *brown* or *bread* by themselves would be a match unless *quick** precedes them.

APIs for search-as-you-type

Follow these links for the REST and .NET SDK reference pages:

- [Suggestions REST API](#)
- [Autocomplete REST API](#)
- [SuggestAsync method](#)
- [AutocompleteAsync method](#)

Structure a response

Responses for autocomplete and suggestions are what you might expect for the pattern: [Autocomplete](#) returns a list of terms, [Suggestions](#) returns terms plus a document ID so that you can fetch the document (use the [Lookup Document API](#) to fetch the specific document for a detail page).

Responses are shaped by the parameters on the request:

- For autocomplete, set the `autocompleteMode` to determine whether text completion occurs on one or two terms.

- For suggestions, set `$select` to return fields containing unique or differentiating values, such as names and description. Avoid fields that contain duplicate values (such as a category or city).

The following parameters apply to both autocomplete and suggestions, but are more applicable to suggestions, especially when a suggester includes multiple fields.

 [Expand table](#)

Parameter	Usage
searchFields	Constrain the query to specific fields.
\$filter	Apply match criteria on the result set (<code>\$filter=Category eq 'ActionAdventure'</code>).
\$top	Limit the results to a specific number (<code>\$top=5</code>).

Add user interaction code

Autofilling a query term or dropping down a list of matching links requires user interaction code, typically JavaScript, that can consume requests from external sources, such as autocomplete or suggestion queries against an Azure Search Cognitive index.

Although you could write this code natively, it's easier to use functions from existing JavaScript library, such as one of the following.

- [Autocomplete widget \(jQuery UI\)](#) appears in the suggestion code snippet. You can create a search box, and then reference it in a JavaScript function that uses the autocomplete widget. Properties on the widget set the source (an autocomplete or suggestions function), minimum length of input characters before action is taken, and positioning.
- [XDSOFT Autocomplete plug-in](#) appears in the autocomplete code snippet.
- [Suggestions](#) appears in the [Add search to web sites tutorial](#) and code sample.

Use these libraries in the client to create a search box that supports both suggestions and autocomplete. Inputs collected in the search box can then be paired with suggestions and autocomplete actions on the search service.

Suggestions

This section walks you through an implementation of suggested results, starting with the search box definition. It also shows how and script that invokes the first JavaScript

autocomplete library referenced in this article.

Create a search box

Assuming the [jQuery UI Autocomplete library](#) and an MVC project in C#, you could define the search box using JavaScript in the *Index.cshtml* file. The library adds the search-as-you-type interaction to the search box by making asynchronous calls to the MVC controller to retrieve suggestions.

In *Index.cshtml* inside the folder *\Views\Home*, a line to create a search box might be as follows:

HTML

```
<input class="searchBox" type="text" id="searchbox1" placeholder="search">
```

This example is a simple input text box with a class for styling, an ID to be referenced by JavaScript, and placeholder text.

Within the same file, embed JavaScript that references the search box. The following function calls the Suggest API, which requests suggested matching documents based on partial term inputs:

JavaScript

```
$(function () {
    $("#searchbox1").autocomplete({
        source: "/home/suggest?highlights=false&fuzzy=false&",
        minLength: 3,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        }
    });
});
```

The `source` tells the jQuery UI Autocomplete function where to get the list of items to show under the search box. Since this project is an MVC project, it calls the `Suggest` function in *HomeController.cs* that contains the logic for returning query suggestions. This function also passes a few parameters to control highlights, fuzzy matching, and term. The autocomplete JavaScript API adds the term parameter.

The `minLength: 3` ensures that recommendations are only shown when there are at least three characters in the search box.

Enable fuzzy matching

Fuzzy search allows you to get results based on close matches even if the user misspells a word in the search box. The edit distance is 1, which means there can be a maximum discrepancy of one character between the user input and a match.

```
JavaScript
```

```
source: "/home/suggest?highlights=false&fuzzy=true&" ,
```

Enable highlighting

Highlighting applies font style to the characters in the result that correspond to the input. For example, if the partial input is *micro*, the result would appear as **microsoft**, **microscope**, and so forth. Highlighting is based on the `HighlightPreTag` and `HighlightPostTag` parameters, defined inline with the `Suggest` function.

```
JavaScript
```

```
source: "/home/suggest?highlights=true&fuzzy=true&" ,
```

Suggest function

If you're using C# and an MVC application, the `HomeController.cs` file in the `Controllers` directory is where you might create a class for suggested results. In .NET, a `Suggest` function is based on the [SuggestAsync method](#). For more information about the .NET SDK, see [How to use Azure AI Search from a .NET Application](#).

The `InitSearch` method creates an authenticated HTTP index client to the Azure AI Search service. Properties on the `SuggestOptions` class determine which fields are searched and returned in the results, the number of matches, and whether fuzzy matching is used.

For autocomplete, fuzzy matching is limited to one edit distance (one omitted or misplaced character). Fuzzy matching in autocomplete queries can sometimes produce unexpected results depending on index size and [how it's sharded](#).

```
C#
```

```
public async Task<ActionResult> SuggestAsync(bool highlights, bool fuzzy, string term)
{
    InitSearch();
```

```

var options = new SuggestOptions()
{
    UseFuzzyMatching = fuzzy,
    Size = 8,
};

if (highlights)
{
    options.HighlightPreTag = "<b>";
    options.HighlightPostTag = "</b>";
}

// Only one suggester can be specified per index.
// The suggester for the Hotels index enables autocomplete/suggestions on the
HotelName field only.
// During indexing, HotelNames are indexed in patterns that support
autocomplete and suggested results.
var suggestResult = await _searchClient.SuggestAsync<Hotel>(term, "sg",
options).ConfigureAwait(false);

// Convert the suggest query results to a list that can be displayed in the
client.
List<string> suggestions = suggestResult.Value.Results.Select(x =>
x.Text).ToList();

// Return the list of suggestions.
return new JsonResult(suggestions);
}

```

The `SuggestAsync` function takes two parameters that determine whether hit highlights are returned or fuzzy matching is used in addition to the search term input. Up to eight matches can be included in suggested results. The method creates a `SuggestOptions object`, which is then passed to the Suggest API. The result is then converted to JSON so it can be shown in the client.

Autocomplete

So far, the search UX code has been centered on suggestions. The next code block shows autocomplete, using the XDSOFT jQuery UI Autocomplete function, passing in a request for Azure AI Search autocomplete. As with the suggestions, in a C# application, code that supports user interaction goes in *index.cshtml*.

JavaScript

```

$(function () {
    // using modified jQuery Autocomplete plugin v1.2.8
    https://xdsoft.net/jqplugins/autocomplete/
    // $.autocomplete -> $.autocompleteInline
    $("#searchbox1").autocompleteInline({

```

```

appendMethod: "replace",
source: [
    function (text, add) {
        if (!text) {
            return;
        }

        $.getJSON("/home/autocomplete?term=" + text, function (data) {
            if (data && data.length > 0) {
                currentSuggestion2 = data[0];
                add(data);
            }
        });
    }
],
});

// complete on TAB and clear on ESC
$("#searchbox1").keydown(function (evt) {
    if (evt.keyCode === 9 /* TAB */ && currentSuggestion2) {
        $("#searchbox1").val(currentSuggestion2);
        return false;
    } else if (evt.keyCode === 27 /* ESC */) {
        currentSuggestion2 = "";
        $("#searchbox1").val("");
    }
});
});

```

Autocomplete function

Autocomplete is based on the [AutocompleteAsync method](#). As with suggestions, this code block would go in the *HomeController.cs* file.

C#

```

public async Task<ActionResult> AutoCompleteAsync(string term)
{
    InitSearch();

    // Setup the autocomplete parameters.
    var ap = new AutocompleteOptions()
    {
        Mode = AutocompleteMode.OneTermWithContext,
        Size = 6
    };
    var autocompleteResult = await _searchClient.AutocompleteAsync(term, "sg",
ap).ConfigureAwait(false);

    // Convert the autocompleteResult results to a list that can be displayed in
    // the client.
    List<string> autocomplete = autocompleteResult.Value.Results.Select(x =>

```

```
x.Text).ToList();  
  
    return new JsonResult(autocomplete);  
}
```

The Autocomplete function takes the search term input. The method creates an [AutoCompleteParameters object](#). The result is then converted to JSON so it can be shown in the client.

Next step

The following tutorial demonstrates a search-as-you-type experience.

[Add search to a web site \(C#\)](#)

Examples of *simple* search queries in Azure AI Search

Article • 04/14/2025

In Azure AI Search, the [simple query syntax](#) invokes the default query parser for full text search. The parser is fast and handles common scenarios, including full text search, filtered and faceted search, and prefix search. This article uses examples to illustrate simple syntax usage in a [Search Documents \(REST API\)](#) request.

ⓘ Note

An alternative query syntax is [Lucene](#), which supports more complex query structures, such as fuzzy and wildcard search. For more information, see [Examples of full Lucene search syntax](#).

Hotels sample index

The following queries are based on the hotels-sample-index, which you can create by following the instructions in [Quickstart: Create a search index in the Azure portal](#).

Example queries are articulated using the REST API and POST requests. You can paste and run them in a [REST client](#). Or, use the JSON view of [Search explorer](#) in the Azure portal. In JSON view, you can paste in the query examples shown here in this article.

Request headers must have the following values:

[+] [Expand table](#)

Key	Value
Content-Type	application/json
api-key	<your-search-service-api-key>, either query or admin key

URI parameters must include your search service endpoint with the index name, docs collections, search command, and API version, similar to the following example:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2024-07-01
```

The request body should be formed as valid JSON:

JSON

```
{  
  "search": "*",  
  "queryType": "simple",  
  "select": "HotelId, HotelName, Category, Tags, Description",  
  "count": true  
}
```

- `search` set to `*` is an unspecified query, equivalent to null or empty search. It's not especially useful, but it's the simplest search you can do, and it shows all retrievable fields in the index, with all values.
- `queryType` set to `simple` is the default and can be omitted, but it's included to emphasize that the query examples in this article are expressed in the simple syntax.
- `select` set to a comma-delimited list of fields is used for search result composition, including just those fields that are useful in the context of search results.
- `count` returns the number of documents matching the search criteria. On an empty search string, the count is all documents in the index (50 in the hotels-sample-index).

Example 1: Full text search

Full text search can be any number of standalone terms or quote-enclosed phrases, with or without Boolean operators.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01  
{  
  "search": "pool spa +airport",  
  "searchMode": "any",  
  "queryType": "simple",  
  "select": "HotelId, HotelName, Category, Description",  
  "count": true  
}
```

A keyword search that's composed of important terms or phrases tend to work best. String fields undergo text analysis during indexing and querying, dropping nonessential words like `the`, `and`, `it`. To see how a query string is tokenized in the index, pass the string in an [Analyze Text](#) call to the index.

The `searchMode` parameter controls precision and recall. If you want more recall, use the default `any` value, which returns a result if any part of the query string is matched. If you favor precision, where all parts of the string must be matched, change `searchMode` to `all`. Try the preceding query both ways to see how `searchMode` changes the outcome.

The response for the `pool spa +airport` query should look similar to the following example.

JSON

```
"@odata.count": 4,
"value": [
{
    "@search.score": 6.090657,
    "HotelId": "12",
    "HotelName": "Winter Panorama Resort",
    "Description": "Plenty of great skiing, outdoor ice skating, sleigh rides, tubing and snow biking. Yoga, group exercise classes and outdoor hockey are available year-round, plus numerous options for shopping as well as great spa services. Newly-renovated with large rooms, free 24-hr airport shuttle & a new restaurant. Rooms/suites offer mini-fridges & 49-inch HDTVs.",
    "Category": "Resort and Spa"
},
{
    "@search.score": 4.314683,
    "HotelId": "21",
    "HotelName": "Good Business Hotel",
    "Description": "1 Mile from the airport. Free WiFi, Outdoor Pool, Complimentary Airport Shuttle, 6 miles from Lake Lanier & 10 miles from downtown. Our business center includes printers, a copy machine, fax, and a work area.",
    "Category": "Suite"
},
{
    "@search.score": 3.575948,
    "HotelId": "27",
    "HotelName": "Starlight Suites",
    "Description": "Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor Outdoor Pool, Fitness Center, Florida Green certified, Complimentary Coffee, HDTV",
    "Category": "Suite"
},
{
    "@search.score": 2.6926985,
    "HotelId": "25",
    "HotelName": "Waterfront Scottish Inn",
    "Description": "Newly Redesigned Rooms & airport shuttle. Minutes from the airport, enjoy lakeside amenities, a resort-style pool & stylish new guestrooms with Internet TVs.",
    "Category": "Suite"
}
]
```

Notice the search score in the response. This is the relevance score of the match. By default, a search service returns the top 50 matches based on this score.

Uniform scores of *1.0* occur when there's no rank, either because the search wasn't full text search, or because no criteria were provided. For example, in an empty search (`search=*`), rows come back in arbitrary order. When you include actual criteria, you'll see search scores evolve into meaningful values.

Example 2: Look up by ID

After search results are returned, a logical next step is to provide a details page that includes more fields from the document. This example shows you how to return a single document using [Get Document](#) by passing in the document ID.

HTTP

```
GET /indexes/hotels-sample-index/docs/41?api-version=2024-07-01
```

All documents have a unique identifier. If you're using the Azure portal, select the index from the **Indexes** tab and then look at the field definitions to determine which field is the key. In the REST API, the [GET Index](#) call returns the index definition in the response body.

The response for the preceding query consists of the document whose key is *41*. Any field that is marked as *retrievable* in the index definition can be returned in search results and rendered in your app.

JSON

```
{
    "HotelId": "41",
    "HotelName": "Windy Ocean Motel",
    "Description": "Oceanfront hotel overlooking the beach features rooms with a private balcony and 2 indoor and outdoor pools. Inspired by the natural beauty of the island, each room includes an original painting of local scenes by the owner. Rooms include a mini fridge, Keurig coffee maker, and flatscreen TV. Various shops and art entertainment are on the boardwalk, just steps away.",
    "Description_fr": "Cet hôtel en bord de mer donnant sur la plage propose des chambres dotées d'un balcon privé et de 2 piscines intérieure et extérieure. Inspiré par la beauté naturelle de l'île, chaque chambre comprend une peinture originale de scènes locales par le propriétaire. Les chambres comprennent un mini-réfrigérateur, une cafetière Keurig et une télévision à écran plat. Divers magasins et divertissements artistiques se trouvent sur la promenade, à quelques pas.",
    "Category": "Suite",
    "Tags": [
        "pool",
        "air conditioning",
```

```

    "bar",
  ],
  "ParkingIncluded": true,
  "LastRenovationDate": "2021-05-10T00:00:00Z",
  "Rating": 3.5,
  "Location": {
    "type": "Point",
    "coordinates": [
      -157.846817,
      21.295841
    ],
    "crs": {
      "type": "name",
      "properties": {
        "name": "EPSG:4326"
      }
    }
  },
  "Address": {
    "StreetAddress": "1450 Ala Moana Blvd 2238 Ala Moana Ctr",
    "City": "Honolulu",
    "StateProvince": "HI",
    "PostalCode": "96814",
    "Country": "USA"
  }
}

```

Example 3: Filter on text

[Filter syntax](#) is an OData expression that you can use by itself or with `search`. When used together in the same request, `filter` is applied first to the entire index, and then the `search` is performed on the results of the filter. Filters can therefore be a useful technique to improve query performance since they reduce the set of documents that the search query needs to process.

Filters can be defined on any field marked as `filterable` in the index definition. For `hotels-sample-index`, filterable fields include `Category`, `Tags`, `ParkingIncluded`, `Rating`, and most `Address` fields.

HTTP

```

POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "art tours",
  "queryType": "simple",
  "filter": "Category eq 'Boutique'",
  "searchFields": "HotelName,Description,Category",
  "select": "HotelId,HotelName,Description,Category",

```

```
    "count": true
}
```

The response for the preceding query is scoped to only those hotels categorized as *Boutique*, and that include the terms *art* or *tours*. In this case, there's just one match.

JSON

```
"value": [
{
  "@search.score": 1.2814453,
  "HotelId": "2",
  "HotelName": "Old Century Hotel",
  "Description": "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.",
  "Category": "Boutique"
}
]
```

Example 4: Filter functions

Filter expressions can include [search.ismatch](#) and [search.ismatchscoring](#) functions, allowing you to build a search query within the filter. This filter expression uses a wildcard on *free* to select amenities including free wifi, free parking, and so forth.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "",
  "filter": "search.ismatch('free*', 'Tags', 'full', 'any')",
  "select": "HotelName, Tags, Description",
  "count": true
}
```

The response for the preceding query matches on 27 hotels that offer free amenities. Notice that the search score is a uniform *1* throughout the results. This is because the search expression is null or empty, resulting in verbatim filter matches, but no full text search. Relevance scores are only returned on full text search. If you're using filters without `search`, make sure you have sufficient sortable fields so that you can control search rank.

JSON

```

"@odata.count": 27,
"value": [
  {
    "@search.score": 1,
    "HotelName": "Country Residence Hotel",
    "Description": "All of the suites feature full-sized kitchens stocked with cookware, separate living and sleeping areas and sofa beds. Some of the larger rooms have fireplaces and patios or balconies. Experience real country hospitality in the heart of bustling Nashville. The most vibrant music scene in the world is just outside your front door.",
    "Tags": [
      "laundry service",
      "restaurant",
      "free parking"
    ]
  },
  {
    "@search.score": 1,
    "HotelName": "Downtown Mix Hotel",
    "Description": "Mix and mingle in the heart of the city. Shop and dine, mix and mingle in the heart of downtown, where fab lake views unite with a cheeky design.",
    "Tags": [
      "air conditioning",
      "laundry service",
      "free wifi"
    ]
  },
  {
    "@search.score": 1,
    "HotelName": "Starlight Suites",
    "Description": "Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor Outdoor Pool, Fitness Center, Florida Green certified, Complimentary Coffee, HDTV",
    "Tags": [
      "pool",
      "coffee in lobby",
      "free wifi"
    ]
  },
  ...

```

Example 5: Range filters

Range filtering is supported through filters expressions for any data type. The following examples illustrate numeric and string ranges. Data types are important in range filters and work best when numeric data is in numeric fields, and string data in string fields. Numeric data in string fields isn't suitable for ranges because numeric strings aren't comparable.

The following query is a numeric range. In hotels-sample-index, the only filterable numeric field is `Rating`.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "*",
  "filter": "Rating ge 2 and Rating lt 4",
  "select": "HotelId, HotelName, Rating",
  "orderby": "Rating desc",
  "count": true
}
```

The response for this query should look similar to the following example, trimmed for brevity.

JSON

```
"@odata.count": 27,
"value": [
  {
    "@search.score": 1,
    "HotelId": "22",
    "HotelName": "Lion's Den Inn",
    "Rating": 3.9
  },
  {
    "@search.score": 1,
    "HotelId": "25",
    "HotelName": "Waterfront Scottish Inn",
    "Rating": 3.8
  },
  {
    "@search.score": 1,
    "HotelId": "2",
    "HotelName": "Old Century Hotel",
    "Rating": 3.6
  },
  ...
]
```

The next query is a range filter over a string field (`Address/StateProvince`):

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "*",
  "filter": "Address/StateProvince ge 'A*' and Address/StateProvince lt 'D*'",
  "select": "HotelId, HotelName, Address/StateProvince",
```

```
    "count": true
}
```

The response for this query should look similar to the following example, trimmed for brevity. In this example, it's not possible to sort by `StateProvince` because the field isn't attributed as *sortable* in the index definition.

JSON

```
{
  "@odata.count": 9,
  "value": [
    {
      "@search.score": 1,
      "HotelId": "39",
      "HotelName": "White Mountain Lodge & Suites",
      "Address": {
        "StateProvince": "CO"
      }
    },
    {
      "@search.score": 1,
      "HotelId": "9",
      "HotelName": "Smile Up Hotel",
      "Address": {
        "StateProvince": "CA "
      }
    },
    {
      "@search.score": 1,
      "HotelId": "7",
      "HotelName": "Roach Motel",
      "Address": {
        "StateProvince": "CA "
      }
    },
    {
      "@search.score": 1,
      "HotelId": "34",
      "HotelName": "Lakefront Captain Inn",
      "Address": {
        "StateProvince": "CT"
      }
    },
    {
      "@search.score": 1,
      "HotelId": "37",
      "HotelName": "Campus Commander Hotel",
      "Address": {
        "StateProvince": "CA "
      }
    }
  ]
}
```

```
 },  
 . . .
```

Example 6: Geospatial search

The hotels-sample-index includes a *Location* field with latitude and longitude coordinates. This example uses the [geo.distance function](#) that filters on documents within the circumference of a starting point, out to an arbitrary distance (in kilometers) that you provide. You can adjust the last value in the query (10) to reduce or enlarge the surface area of the query.

HTTP

```
POST /indexes/v/docs/search?api-version=2024-07-01  
{  
  "search": "*",  
  "filter": "geo.distance(Location, geography'POINT(-122.335114 47.612839)' ) le  
10",  
  "select": "HotelId, HotelName, Address/City, Address/StateProvince",  
  "count": true  
}
```

The response for this query returns all hotels within a 10-kilometer distance of the coordinates provided:

JSON

```
{  
  "@odata.count": 3,  
  "value": [  
    {  
      "@search.score": 1,  
      "HotelId": "45",  
      "HotelName": "Happy Lake Resort & Restaurant",  
      "Address": {  
        "City": "Seattle",  
        "StateProvince": "WA"  
      }  
    },  
    {  
      "@search.score": 1,  
      "HotelId": "24",  
      "HotelName": "Uptown Chic Hotel",  
      "Address": {  
        "City": "Seattle",  
        "StateProvince": "WA"  
      }  
    },  
    {  
      "@search.score": 1,
```

```
        "HotelId": "16",
        "HotelName": "Double Sanctuary Resort",
        "Address": {
            "City": "Seattle",
            "StateProvince": "WA"
        }
    }
]
```

Example 7: Booleans with searchMode

Simple syntax supports Boolean operators in the form of characters (+, -, |) to support AND, OR, and NOT query logic. Boolean search behaves as you might expect, with a few noteworthy exceptions.

In a Boolean search, consider adding the `searchMode` parameter as a mechanism for influencing precision and recall. Valid values include `"searchMode": "any"` favoring recall (a document that satisfies any of the criteria is considered a match), and `"searchMode": "all"` favoring precision (all criteria must be matched in a document).

In the context of a Boolean search, the default `"searchMode": "any"` can be confusing if you're stacking a query with multiple operators and getting broader instead of narrower results. This is particularly true with NOT, where results include all documents *not containing* a specific term or phrase.

The following example provides an illustration. The query looks for matches on *restaurant* that exclude the phrase *air conditioning*. If you run the following query with `searchMode` (any), 43 documents are returned: those containing the term *restaurant*, plus all documents that *don't* have the phrase **air conditioning*.

Notice that there's no space between the boolean operator (-) and the phrase *air conditioning*. The quotation marks are escaped (\").

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
    "search": "restaurant -\"air conditioning\",
    "searchMode": "any",
    "searchFields": "Tags",
    "select": "HotelId, HotelName, Tags",
    "count": true
}
```

Changing to `"searchMode": "all"` enforces a cumulative effect on criteria and returns a smaller result set (seven matches) consisting of documents containing the term *restaurant*, minus those containing the phrase *air conditioning*.

The response for this query would now look similar to the following example, trimmed for brevity.

```
JSON

{
  "@odata.count": 14,
  "value": [
    {
      "@search.score": 3.1383743,
      "HotelId": "18",
      "HotelName": "Ocean Water Resort & Spa",
      "Tags": [
        "view",
        "pool",
        "restaurant"
      ],
    },
    {
      "@search.score": 2.028083,
      "HotelId": "22",
      "HotelName": "Lion's Den Inn",
      "Tags": [
        "laundry service",
        "free wifi",
        "restaurant"
      ],
    },
    {
      "@search.score": 2.028083,
      "HotelId": "34",
      "HotelName": "Lakefront Captain Inn",
      "Tags": [
        "restaurant",
        "laundry service",
        "coffee in lobby"
      ],
    },
    ...
  ]
}
```

Example 8: Paging results

In previous examples, you learned about parameters that affect search results composition, including `select` that determines which fields are in a result, sort orders, and how to include a count of all matches. This example is a continuation of search result composition in the form of

paging parameters that allow you to batch the number of results that appear in any given page.

By default, a search service returns the top 50 matches. To control the number of matches in each page, use `top` to define the size of the batch, and then use `skip` to pick up subsequent batches.

The following example uses a filter and sort order on the `Rating` field (Rating is both filterable and sortable) because it's easier to see the effects of paging on sorted results. In a regular full search query, the top matches are ranked and paged by `@search.score`.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "*",
  "filter": "Rating gt 4",
  "select": "HotelName, Rating",
  "orderby": "Rating desc",
  "top": 5,
  "count": true
}
```

The query finds 21 matching documents, but because you specified `top`, the response returns just the top five matches, with ratings starting at 4.9, and ending at 4.7 with *Lakeside B & B*.

To get the next five, skip the first batch:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "*",
  "filter": "Rating gt 4",
  "select": "HotelName, Rating",
  "orderby": "Rating desc",
  "top": 5,
  "skip": 5,
  "count": true
}
```

The response for the second batch skips the first five matches, returning the next five, starting with *Pull'r Inn Motel*. To continue with more batches, you would keep `top` at five, and then increment `skip` by five on each new request (`skip=5`, `skip=10`, `skip=15`, and so forth).

JSON

```
{  
    "@odata.count": 21,  
    "value": [  
        {  
            "@search.score": 1,  
            "HotelName": "Head Wind Resort",  
            "Rating": 4.7  
        },  
        {  
            "@search.score": 1,  
            "HotelName": "Sublime Palace Hotel",  
            "Rating": 4.6  
        },  
        {  
            "@search.score": 1,  
            "HotelName": "City Skyline Antiquity Hotel",  
            "Rating": 4.5  
        },  
        {  
            "@search.score": 1,  
            "HotelName": "Nordick's Valley Motel",  
            "Rating": 4.5  
        },  
        {  
            "@search.score": 1,  
            "HotelName": "Winter Panorama Resort",  
            "Rating": 4.5  
        }  
    ]  
}
```

Related content

Now that you have some practice with the basic query syntax, try specifying queries in code. The following link covers how to set up search queries using the Azure SDKs.

- [Quickstart: Full text search using the Azure SDKs](#)

More syntax reference, query architecture, and examples can be found in the following links:

- [Examples of full Lucene search syntax](#)
- [Full text search in Azure AI Search](#)
- [Simple query syntax in Azure AI Search](#)
- [Lucene query syntax in Azure AI Search](#)
- [OData \\$filter syntax in Azure AI Search](#)

Add spell check to queries in Azure AI Search

08/27/2025

ⓘ Important

Spell correction is in public preview under [supplemental terms of use](#). It's available through the Azure portal, preview REST APIs, and beta versions of Azure SDK libraries.

You can improve recall by spell-correcting words in a query before they reach the search engine. The `speller` parameter is supported for all text (non-vector) query types.

Prerequisites

- A search service at the Basic tier or higher, in any region.
- An existing search index with content in a [supported language](#).
- [A query request](#) that has `speller=lexicon` and `queryLanguage` set to a [supported language](#). Spell check works on strings passed in the `search` parameter. It's not supported for filters, fuzzy search, wildcard search, regular expressions, or vector queries.

Use a search client that supports preview APIs on the query request. You can use a [REST client](#) or beta releases of the Azure SDKs.

[] [Expand table](#)

Client library	Versions
REST API	Versions 2020-06-30-Preview and later. We recommend the latest preview API: 2025-08-01-preview
Azure SDK for .NET	version 11.7.0-beta.4
Azure SDK for Java	version 11.8.0-beta.7
Azure SDK for JavaScript	version 11.3.0-beta.8
Azure SDK for Python	version 11.6.0b12

Spell correction with simple search

The following example uses the built-in hotels-sample index to demonstrate spell correction on a simple text query. Without spell correction, the query returns zero results. With correction, the query returns one result for Johnson's family-oriented resort.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-08-01-preview
{
    "search": "famly acitvites",
    "speller": "lexicon",
    "queryLanguage": "en-us",
    "queryType": "simple",
    "select": "HotelId,HotelName,Description,Category,Tags",
    "count": true
}
```

Spell correction with full Lucene

Spelling correction occurs on individual query terms that undergo text analysis, which is why you can use the speller parameter with some Lucene queries, but not others.

- Incompatible query forms that bypass text analysis include: wildcard, regex, fuzzy
- Compatible query forms include: fielded search, proximity, term boosting

This example uses fielded search over the Category field, with full Lucene syntax, and a misspelled query term. By including speller, the typo in "Suiite" is corrected and the query succeeds.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-08-01-preview
{
    "search": "Category:(Resort and Spa) OR Category:Suiite",
    "queryType": "full",
    "speller": "lexicon",
    "queryLanguage": "en-us",
    "select": "Category",
    "count": true
}
```

Spell correction with semantic ranking

This query, with typos in every term except one, undergoes spelling corrections to return relevant results. To learn more, see [Configure semantic ranker](#).

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-08-01-preview
{
    "search": "hisotoric hotell wiht great restraint nad wiifi",
    "queryType": "semantic",
    "speller": "lexicon",
    "queryLanguage": "en-us",
    "searchFields": "HotelName,Tags,Description",
    "select": "HotelId,HotelName,Description,Category,Tags",
    "count": true
}
```

Supported languages

Valid values for `queryLanguage` can be found in the following table, copied from the list of [supported languages \(REST API reference\)](#).

[] Expand table

Language	queryLanguage
English [EN]	EN, EN-US (default)
Spanish [ES]	ES, ES-ES (default)
French [FR]	FR, FR-FR (default)
German [DE]	DE, DE-DE (default)
Dutch [NL]	NL, NL-BE, NL-NL (default)

!Note

Previously, while semantic ranker was in public preview, the `queryLanguage` parameter was also used for semantic ranking. Semantic ranker is now language-agnostic.

Language analyzer considerations

Indexes that contain non-English content often use [language analyzers](#) on non-English fields to apply the linguistic rules of the native language.

When adding spell check to content that also undergoes language analysis, you can achieve better results using the same language for each indexing and query processing step. For example, if a field's content was indexed using the "fr.microsoft" language analyzer, then queries and spell check should all use a French lexicon or language library of some form.

To recap how language libraries are used in Azure AI Search:

- Language analyzers can be invoked during indexing and query execution, and are either Apache Lucene (for example, "de.lucene") or Microsoft ("de.microsoft").
- Language lexicons invoked during spell check are specified using one of the language codes in the [supported language table](#).

In a query request, the value assigned to `queryLanguage` applies to `speller`.

 **Note**

Language consistency across various property values is only a concern if you are using language analyzers. If you are using language-agnostic analyzers (such as keyword, simple, standard, stop, whitespace, or `standardasciifolding.lucene`), then the `queryLanguage` value can be whatever you want.

While content in a search index can be composed in multiple languages, the query input is most likely in one. The search engine doesn't check for compatibility of `queryLanguage`, language analyzer, and the language in which content is composed, so be sure to scope queries accordingly to avoid producing incorrect results.

Next steps

- [Create a basic query](#)
- [Use full Lucene query syntax](#)
- [Use simple query syntax](#)

Filters in keyword search

Article • 03/11/2025

A *filter* provides value-based criteria for including or excluding content before query execution for keyword search, or before or after query execution for vector search.

Filters are applied to nonvector fields, but can be used in vector search if documents include nonvector fields. For example, for indexes organized around chunked content, you might have parent-level fields or metadata fields that can be filtered.

This article explains filtering for keyword search. For more information about vectors, see [Add a filter in a vector query](#).

A filter is specified using [OData filter expression syntax](#). In contrast with keyword and vector search, a filter succeeds only if the match is exact.

When to use a filter

Filters are foundational to several search experiences, including "find near me" geospatial search, faceted navigation, and security filters that show only those documents a user is allowed to see. If you implement any one of these experiences, a filter is required. It's the filter attached to the search query that provides the geolocation coordinates, the facet category selected by the user, or the security ID of the requestor.

Common scenarios include:

- Slice search results based on content in the index. Given a schema with hotel location, categories, and amenities, you might create a filter to explicitly match on criteria (in Seattle, on the water, with a view).
- Implement a search experience comes with a filter dependency:
 - [Faceted navigation](#) uses a filter to pass back the facet category selected by the user.
 - [Geospatial search](#) uses a filter to pass coordinates of the current location in "find near me" apps and functions that match within an area or by distance.
 - [Security filters](#) pass security identifiers as filter criteria, where a match in the index serves as a proxy for access rights to the document.
- Do a "numbers search". Numeric fields are retrievable and can appear in search results, but they aren't searchable (subject to full text search) individually. If you need selection criteria based on numeric data, use a filter.

How filters are executed

At query time, a filter parser accepts criteria as input, converts the expression into atomic Boolean expressions represented as a tree, and then evaluates the filter tree over filterable fields in an index.

Filtering occurs in tandem with search, qualifying which documents to include in downstream processing for document retrieval and relevance scoring. When paired with a search string, the filter effectively reduces the recall set of the subsequent search operation. When used alone (for example, when the query string is empty where `search=*`), the filter criteria is the sole input.

How filters are defined

Filters apply to text and numeric (nonvector) content on fields that are attributed as `filterable`.

Filters are OData expressions, articulated in the [filter syntax](#) supported by Azure AI Search.

You can specify one filter for each `search` operation, but the filter itself can include multiple fields, multiple criteria, and if you use an `ismatch` function, multiple full-text search expressions. In a multi-part filter expression, you can specify predicates in any order (subject to the rules of operator precedence). There's no appreciable gain in performance if you try to rearrange predicates in a particular sequence.

One of the limits on a filter expression is the maximum size limit of the request. The entire request, inclusive of the filter, can be a maximum of 16 MB for POST, or 8 KB for GET. There's also a limit on the number of clauses in your filter expression. A good rule of thumb is that if you have hundreds of clauses, you are at risk of running into the limit. We recommend designing your application in such a way that it doesn't generate filters of unbounded size.

The following examples represent prototypical filter definitions in several APIs.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?  
api-version=2024-07-01  
{  
    "search": "*",  
    "filter": "Rooms/any(room: room/BaseRate lt 150.0)",
```

```
        "select": "HotelId, HotelName, Rooms/Description, Rooms/BaseRate"  
    }  
  
C#
```

```
options = new SearchOptions()  
{  
    Filter = "Rating gt 4",  
    OrderBy = { "Rating desc" }  
};
```

Filter patterns

The following examples illustrate several usage patterns for filter scenarios. For more ideas, see [OData expression syntax > Examples](#).

- Standalone **\$filter**, without a query string, useful when the filter expression is able to fully qualify documents of interest. Without a query string, there's no lexical or linguistic analysis, no scoring, and no ranking. Notice the search string is just an asterisk, which means "match all documents".

JSON

```
{  
    "search": "*",  
    "filter": "Rooms/any(room: room/BaseRate ge 60 and room/BaseRate lt  
300) and Address/City eq 'Honolulu'"  
}
```

- Combination of query string and **\$filter**, where the filter creates the subset, and the query string provides the term inputs for full text search over the filtered subset. The addition of terms (walking distance theaters) introduces search scores in the results, where documents that best match the terms are ranked higher. Using a filter with a query string is the most common usage pattern.

JSON

```
{  
    "search": "walking distance theaters",  
    "filter": "Rooms/any(room: room/BaseRate ge 60 and room/BaseRate lt  
300) and Address/City eq 'Seattle'"  
}
```

- Compound queries, separated by "or", each with its own filter criteria (for example, 'beagles' in 'dog' or 'siamese' in 'cat'). Expressions combined with `or` are evaluated individually, with the union of documents matching each expression sent back in the response. This usage pattern is achieved through the `search.ismatchscoring` function. You can also use the nonscoring version, `search.ismatch`.

HTTP

```
# Match on hostels rated higher than 4 OR 5-star motels.
$filter=search.ismatchscoring('hostel') and Rating ge 4 or
search.ismatchscoring('motel') and Rating eq 5

# Match on 'luxury' or 'high-end' in the description field OR on
category exactly equal to 'Luxury'.
$filter=search.ismatchscoring('luxury | high-end', 'Description') or
Category eq 'Luxury'&$count=true
```

It's also possible to combine full-text search via `search.ismatchscoring` with filters using `and` instead of `or`, but this is functionally equivalent to using the `search` and `$filter` parameters in a search request. For example, the following two queries produce the same result:

HTTP

```
$filter=search.ismatchscoring('pool') and Rating ge 4

search=pool&$filter=Rating ge 4
```

Field requirements for filtering

In the REST API, filterable is *on* by default for simple fields. Filterable fields increase index size; be sure to set `"filterable": false` for fields that you don't plan to actually use in a filter. For more information about settings for field definitions, see [Create Index](#).

In the Azure SDKs, filterable is *off* by default. You can make a field filterable by setting the [IsFilterable property](#) of the corresponding [SearchField](#) object to `true`. In the next example, the attribute is set on the `Rating` property of a model class that maps to the index definition.

C#

```
[SearchField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
public double? Rating { get; set; }
```

Making an existing field filterable

You can't modify existing fields to make them filterable. Instead, you need to add a new field, or rebuild the index. For more information about rebuilding an index or repopulating fields, see [How to rebuild an Azure AI Search index](#).

Text filter fundamentals

Text filters match string fields against literal strings that you provide in the filter:

```
$filter=Category eq 'Resort and Spa'
```

Unlike full-text search, there's no lexical analysis or word-breaking for text filters, so comparisons are for exact matches only. For example, assume a field *f* contains "sunny day", `$filter=f eq 'sunny'` doesn't match, but `$filter=f eq 'sunny day'` will.

Text strings are case-sensitive, which means text filters are case sensitive by default. For example, `$filter=f eq 'Sunny day'` won't find "sunny day". However, you can use a [normalizer](#) to make it so filtering isn't case sensitive.

Approaches for filtering on text

[+] [Expand table](#)

Approach	Description	When to use
<code>search.in</code>	A function that matches a field against a delimited list of strings.	Recommended for security filters and for any filters where many raw text values need to be matched with a string field. The <code>search.in</code> function is designed for speed and is much faster than explicitly comparing the field against each string using <code>eq</code> and <code>or</code> .
<code>search.ismatch</code>	A function that allows you to mix full-text search operations with strictly Boolean filter operations in the same filter expression.	Use <code>search.ismatch</code> (or its scoring equivalent, <code>search.ismatchscoring</code>) when you want multiple search-filter combinations in one request. You can also use it for a <i>contains</i> filter to filter on a partial string within a larger string.
<code>\$filter=field operator string</code>	A user-defined expression composed of fields, operators, and values.	Use this when you want to find exact matches between a string field and a string value.

Numeric filter fundamentals

Numeric fields aren't `searchable` in the context of full text search. Only strings are subject to full text search. For example, if you enter 99.99 as a search term, you won't get back items priced at \$99.99. Instead, you would see items that have the number 99 in string fields of the document. Thus, if you have numeric data, the assumption is that you'll use them for filters, including ranges, facets, groups, and so forth.

Documents that contain numeric fields (price, size, SKU, ID) provide those values in search results if the field is marked `retrievable`. The point here's that full text search itself isn't applicable to numeric field types.

Next steps

First, try **Search explorer** in the Azure portal to submit queries with `$filter` parameters. The [real-estate-sample index](#) provides interesting results for the following filtered queries when you paste them into the search bar:

```
HTTP

# Geo-filter returning documents within 5 kilometers of Redmond, Washington state
# Use $count=true to get a number of hits returned by the query
# Use $select to trim results, showing values for named fields only
# Use search=* for an empty query string. The filter is the sole input

search=*&$count=true&$select=description,city,postCode&$filter=geo.distance(location,geography'POINT(-122.121513 47.673988)') le 5

# Numeric filters use comparison like greater than (gt), less than (lt), not equal (ne)
# Include "and" to filter on multiple fields (baths and bed)
# Full text search is on John Leclerc, matching on John or Leclerc

search=John
Leclerc&$count=true&$select=source,city,postCode,baths,beds&$filter=baths gt 3 and beds gt 4

# Text filters can also use comparison operators
# Wrap text in single or double quotes and use the correct case
# Full text search is on John Leclerc, matching on John or Leclerc

search=John
Leclerc&$count=true&$select=source,city,postCode,baths,beds&$filter=city gt 'Seattle'
```

To work with more examples, see [OData Filter Expression Syntax > Examples](#).

See also

- [How full text search works in Azure AI Search](#)
 - [Search Documents REST API](#)
 - [Simple query syntax](#)
 - [Lucene query syntax](#)
 - [Supported data types](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Understand how OData collection filters work in Azure AI Search

05/29/2025

This article provides background for developers who are writing advanced filters with complex lambda expressions. The article explains why the rules for collection filters exist by exploring how Azure AI Search executes these filters.

When you build a [filter](#) on collection fields in Azure AI Search, you can use the [any and all operators](#) together with [lambda expressions](#). Lambda expressions are Boolean expressions that refer to a [range variable](#). In filters that use a lambda expression, the `any` and `all` operators are analogous to a `for` loop in most programming languages, with the range variable taking the role of loop variable, and the lambda expression as the body of the loop. The range variable takes on the "current" value of the collection during iteration of the loop.

At least that's how it works conceptually. In reality, Azure AI Search implements filters in a very different way to how `for` loops work. Ideally, this difference would be invisible to you, but in certain situations it isn't. The end result is that there are rules you have to follow when writing lambda expressions.

ⓘ Note

For information on what the rules for collection filters are, including examples, see [Troubleshooting OData collection filters in Azure AI Search](#).

Why collection filters are limited

There are three underlying reasons why filter features aren't fully supported for all types of collections:

1. Only certain operators are supported for certain data types. For example, it doesn't make sense to compare the Boolean values `true` and `false` using `lt`, `gt`, and so on.
2. Azure AI Search doesn't support *correlated search* on fields of type `Collection(Edm.ComplexType)`.
3. Azure AI Search uses inverted indexes to execute filters over all types of data, including collections.

The first reason is just a consequence of how the OData language and EDM type system are defined. The last two are explained in more detail in the rest of this article.

Correlated versus uncorrelated search

When you apply multiple filter criteria over a collection of complex objects, the criteria are correlated because they apply to *each object in the collection*. For example, the following filter returns hotels that have at least one deluxe room with a rate less than 100:

odata-filter-expr

```
Rooms/any(room: room/Type eq 'Deluxe Room' and room/BaseRate lt 100)
```

If filtering was *uncorrelated*, the above filter might return hotels where one room is deluxe and a different room has a base rate less than 100. That wouldn't make sense, since both clauses of the lambda expression apply to the same range variable, namely `room`. This is why such filters are correlated.

However, for full-text search, there's no way to refer to a specific range variable. If you use fielded search to issue a [full Lucene query](#) like this one:

odata-filter-expr

```
Rooms/Type:deluxe AND Rooms/Description:"city view"
```

you might get hotels back where one room is deluxe, and a different room mentions "city view" in the description. For example, the document below with `Id` of `1` would match the query:

JSON

```
{
  "value": [
    {
      "Id": "1",
      "Rooms": [
        { "Type": "deluxe", "Description": "Large garden view suite" },
        { "Type": "standard", "Description": "Standard city view room" }
      ]
    },
    {
      "Id": "2",
      "Rooms": [
        { "Type": "deluxe", "Description": "Courtyard motel room" }
      ]
    }
  ]
}
```

The reason is that `Rooms/Type` refers to all the analyzed terms of the `Rooms/Type` field in the entire document, and similarly for `Rooms/Description`, as shown in the tables below.

How `Rooms/Type` is stored for full-text search:

 Expand table

Term in <code>Rooms/Type</code>	Document IDs
deluxe	1, 2
standard	1

How `Rooms/Description` is stored for full-text search:

 Expand table

Term in <code>Rooms/Description</code>	Document IDs
courtyard	2
city	1
garden	1
large	1
motel	2
room	1, 2
standard	1
suite	1
view	1

So unlike the filter above, which basically says "match documents where a room has `Type` equal to 'Deluxe Room' and **that same room** has `BaseRate` less than 100", the search query says "match documents where `Rooms/Type` has the term "deluxe" and `Rooms/Description` has the phrase "city view". There's no concept of individual rooms whose fields can be correlated in the latter case.

Inverted indexes and collections

You might have noticed that there are far fewer restrictions on lambda expressions over complex collections than there are for simple collections like `Collection(Edm.Int32)`, `Collection(Edm.GeographyPoint)`, and so on. This is because Azure AI Search stores complex collections as actual collections of subdocuments, while simple collections aren't stored as collections at all.

For example, consider a filterable string collection field like `seasons` in an index for an online retailer. Some documents uploaded to this index might look like this:

```
JSON

{
  "value": [
    {
      "id": "1",
      "name": "Hiking boots",
      "seasons": ["spring", "summer", "fall"]
    },
    {
      "id": "2",
      "name": "Rain jacket",
      "seasons": ["spring", "fall", "winter"]
    },
    {
      "id": "3",
      "name": "Parka",
      "seasons": ["winter"]
    }
  ]
}
```

The values of the `seasons` field are stored in a structure called an **inverted index**, which looks something like this:

[Expand table](#)

Term	Document IDs
spring	1, 2
summer	1
fall	1, 2
winter	2, 3

This data structure is designed to answer one question with great speed: In which documents does a given term appear? Answering this question works more like a plain equality check than

a loop over a collection. In fact, this is why for string collections, Azure AI Search only allows `eq` as a comparison operator inside a lambda expression for `any`.

Next, we look at how it's possible to combine multiple equality checks on the same range variable with `or`. It works thanks to algebra and [the distributive property of quantifiers](#). This expression:

```
odata-filter-expr
```

```
seasons/any(s: s eq 'winter' or s eq 'fall')
```

is equivalent to:

```
odata-filter-expr
```

```
seasons/any(s: s eq 'winter') or seasons/any(s: s eq 'fall')
```

and each of the two `any` sub-expressions can be efficiently executed using the inverted index. Also, thanks to [the negation law of quantifiers](#), this expression:

```
odata-filter-expr
```

```
seasons/all(s: s ne 'winter' and s ne 'fall')
```

is equivalent to:

```
odata-filter-expr
```

```
not seasons/any(s: s eq 'winter' or s eq 'fall')
```

which is why it's possible to use `all` with `ne` and `and`.

⚠ Note

Although the details are beyond the scope of this document, these same principles extend to [distance and intersection tests for collections of geo-spatial points](#) as well. This is why, in `any`:

- `geo.intersects` cannot be negated
- `geo.distance` must be compared using `lt` or `le`
- expressions must be combined with `or`, not `and`

The converse rules apply for `all`.

A wider variety of expressions are allowed when filtering on collections of data types that support the `lt`, `gt`, `le`, and `ge` operators, such as `Collection(Edm.Int32)` for example. Specifically, you can use `and` as well as `or` in `any`, as long as the underlying comparison expressions are combined into **range comparisons** using `and`, which are then further combined using `or`. This structure of Boolean expressions is called [Disjunctive Normal Form \(DNF\)](#), otherwise known as "ORs of ANDs". Conversely, lambda expressions for `all` for these data types must be in [Conjunctive Normal Form \(CNF\)](#), otherwise known as "ANDs of ORs". Azure AI Search allows such range comparisons because it can execute them using inverted indexes efficiently, just like it can do fast term lookup for strings.

In summary, here are the rules of thumb for what's allowed in a lambda expression:

- Inside `any`, *positive checks* are always allowed, like equality, range comparisons, `geo.intersects`, or `geo.distance` compared with `lt` or `le` (think of "closeness" as being like equality when it comes to checking distance).
- Inside `any`, `or` is always allowed. You can use `and` only for data types that can express range checks, and only if you use ORs of ANDs (DNF).
- Inside `all`, the rules are reversed. Only *negative checks* are allowed, you can use `and` always, and you can use `or` only for range checks expressed as ANDs of ORs (CNF).

In practice, these are the types of filters you're most likely to use anyway. It's still helpful to understand the boundaries of what's possible though.

For specific examples of which kinds of filters are allowed and which aren't, see [How to write valid collection filters](#).

Next steps

- [Troubleshooting OData collection filters in Azure AI Search](#)
- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Troubleshooting OData collection filters in Azure AI Search

05/29/2025

To [filter](#) on collection fields in Azure AI Search, you can use the [any and all operators](#) together with **lambda expressions**. A lambda expression is a subfilter that is applied to each element of a collection.

Not every feature of filter expressions is available inside a lambda expression. Which features are available differs depending on the data type of the collection field that you want to filter. This can result in an error if you try to use a feature in a lambda expression that isn't supported in that context. If you're encountering such errors while trying to write a complex filter over collection fields, this article will help you troubleshoot the problem.

Common collection filter errors

The following table lists errors that you might encounter when trying to execute a collection filter. These errors happen when you use a feature of filter expressions that isn't supported inside a lambda expression. Each error gives some guidance on how you can rewrite your filter to avoid the error. The table also includes a link to the relevant section of this article that provides more information on how to avoid that error.

[Expand table](#)

Error message	Situation	Details
The function <code>ismatch</code> has no parameters bound to the range variable 's'. Only bound field references are supported inside lambda expressions ('any' or 'all'). However, you can change your filter so that the <code>ismatch</code> function is outside the lambda expression and try again.	Using <code>search.ismatch</code> or <code>search.ismatchscoring</code> inside a lambda expression	Rules for filtering complex collections
Invalid lambda expression. Found a test for equality or inequality where the opposite was expected in a lambda expression that iterates over a field of type Collection(Edm.String). For 'any', use expressions of the form 'x eq y' or 'search.in(...)'. For 'all', use expressions of the form 'x ne y', 'not (x eq y)', or 'not search.in(...)'.	Filtering on a field of type <code>Collection(Edm.String)</code>	Rules for filtering string collections
Invalid lambda expression. Found an unsupported form of complex Boolean	Filtering on fields of type <code>Collection(Edm.DateTimeOffset)</code> ,	Rules for filtering

Error message	Situation	Details
<p>expression. For 'any', use expressions that are 'ORs of ANDs', also known as Disjunctive Normal Form. For example: <code>(a and b) or (c and d)</code> where a, b, c, and d are comparison or equality subexpressions. For 'all', use expressions that are 'ANDs of ORs', also known as Conjunctive Normal Form. For example: <code>(a or b) and (c or d)</code> where a, b, c, and d are comparison or inequality subexpressions.</p> <p>Examples of comparison expressions: 'x gt 5', 'x le 2'. Example of an equality expression: 'x eq 5'. Example of an inequality expression: 'x ne 5'.</p>	<code>Collection(Edm.Double)</code> , <code>Collection(Edm.Int32)</code> , or <code>Collection(Edm.Int64)</code>	comparable collections
<p>Invalid lambda expression. Found an unsupported use of <code>geo.distance()</code> or <code>geo.intersects()</code> in a lambda expression that iterates over a field of type <code>Collection(Edm.GeographyPoint)</code>. For 'any', make sure you compare <code>geo.distance()</code> using the 'lt' or 'le' operators and make sure that any usage of <code>geo.intersects()</code> isn't negated. For 'all', make sure you compare <code>geo.distance()</code> using the 'gt' or 'ge' operators and make sure that any usage of <code>geo.intersects()</code> is negated.</p>	Filtering on a field of type <code>Collection(Edm.GeographyPoint)</code>	Rules for filtering GeographyPoint collections
<p>Invalid lambda expression. Complex Boolean expressions aren't supported in lambda expressions that iterate over fields of type <code>Collection(Edm.GeographyPoint)</code>. For 'any', join subexpressions with 'or'; 'and' isn't supported. For 'all', join subexpressions with 'and'; 'or' isn't supported.</p>	Filtering on fields of type <code>Collection(Edm.String)</code> or <code>Collection(Edm.GeographyPoint)</code>	Rules for filtering string collections
<p>Invalid lambda expression. Found a comparison operator (one of 'lt', 'le', 'gt', or 'ge'). Only equality operators are allowed in lambda expressions that iterate over fields of type <code>Collection(Edm.String)</code>. For 'any', see expressions of the form 'x eq y'. For 'all', use expressions of the form 'x ne y' or 'not (x eq y)'.</p>	Filtering on a field of type <code>Collection(Edm.String)</code>	Rules for filtering string collections

How to write valid collection filters

The rules for writing valid collection filters are different for each data type. The following sections describe the rules by showing examples of which filter features are supported and

which aren't:

- Rules for filtering string collections
- Rules for filtering Boolean collections
- Rules for filtering GeographyPoint collections
- Rules for filtering comparable collections
- Rules for filtering complex collections

Rules for filtering string collections

Inside lambda expressions for string collections, the only comparison operators that can be used are `eq` and `ne`.

(!) Note

Azure AI Search does not support the `lt`/`le`/`gt`/`ge` operators for strings, whether inside or outside a lambda expression.

The body of an `any` can only test for equality while the body of an `all` can only test for inequality.

It's also possible to combine multiple expressions via `or` in the body of an `any`, and via `and` in the body of an `all`. Since the `search.in` function is equivalent to combining equality checks with `or`, it's also allowed in the body of an `any`. Conversely, `not search.in` is allowed in the body of an `all`.

For example, these expressions are allowed:

- `tags/any(t: t eq 'books')`
- `tags/any(t: search.in(t, 'books, games, toys'))`
- `tags/all(t: t ne 'books')`
- `tags/all(t: not (t eq 'books'))`
- `tags/all(t: not search.in(t, 'books, games, toys'))`
- `tags/any(t: t eq 'books' or t eq 'games')`
- `tags/all(t: t ne 'books' and not (t eq 'games'))`

While these expressions aren't allowed:

- `tags/any(t: t ne 'books')`
- `tags/any(t: not search.in(t, 'books, games, toys'))`
- `tags/all(t: t eq 'books')`

- `tags/all(t: search.in(t, 'books, games, toys'))`
- `tags/any(t: t eq 'books' and t ne 'games')`
- `tags/all(t: t ne 'books' or not (t eq 'games'))`

Rules for filtering Boolean collections

The type `Edm.Boolean` supports only the `eq` and `ne` operators. As such, it doesn't make much sense to allow combining such clauses that check the same range variable with `and`/`or` since that would always lead to tautologies or contradictions.

Here are some examples of filters on Boolean collections that are allowed:

- `flags/any(f: f)`
- `flags/all(f: f)`
- `flags/any(f: f eq true)`
- `flags/any(f: f ne true)`
- `flags/all(f: not f)`
- `flags/all(f: not (f eq true))`

Unlike string collections, Boolean collections have no limits on which operator can be used in which type of lambda expression. Both `eq` and `ne` can be used in the body of `any` or `all`.

Expressions such as the following aren't allowed for Boolean collections:

- `flags/any(f: f or not f)`
- `flags/any(f: f or f)`
- `flags/all(f: f and not f)`
- `flags/all(f: f and f eq true)`

Rules for filtering GeographyPoint collections

Values of type `Edm.GeographyPoint` in a collection can't be compared directly to each other. Instead, they must be used as parameters to the `geo.distance` and `geo.intersects` functions. The `geo.distance` function in turn must be compared to a distance value using one of the comparison operators `lt`, `le`, `gt`, or `ge`. These rules also apply to noncollection `Edm.GeographyPoint` fields.

Like string collections, `Edm.GeographyPoint` collections have some rules for how the geo-spatial functions can be used and combined in the different types of lambda expressions:

- Which comparison operators you can use with the `geo.distance` function depends on the type of lambda expression. For `any`, you can use only `lt` or `le`. For `all`, you can use only `gt` or `ge`. You can negate expressions involving `geo.distance`, but you have to change the comparison operator (`geo.distance(...)` `lt` `x` becomes `not (geo.distance(...)` `ge` `x)` and `geo.distance(...)` `le` `x` becomes `not (geo.distance(...)` `gt` `x)).`
- In the body of an `all`, the `geo.intersects` function must be negated. Conversely, in the body of an `any`, the `geo.intersects` function must not be negated.
- In the body of an `any`, geo-spatial expressions can be combined using `or`. In the body of an `all`, such expressions can be combined using `and`.

The above limitations exist for similar reasons as the equality/inequality limitation on string collections. See [Understanding OData collection filters in Azure AI Search](#) for a deeper look at these reasons.

Here are some examples of filters on `Edm.GeographyPoint` collections that are allowed:

- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') lt 10)`
- `locations/any(l: not (geo.distance(l, geography'POINT(-122 49)') ge 10) or geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') ge 10 and not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`

Expressions such as the following aren't allowed for `Edm.GeographyPoint` collections:

- `locations/any(l: l eq geography'POINT(-122 49)')`
- `locations/any(l: not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`
- `locations/all(l: geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`
- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') gt 10)`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') lt 10)`
- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') lt 10 and geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') le 10 or not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581)))')`

Rules for filtering comparable collections

This section applies to all the following data types:

- `Collection(Edm.DateTimeOffset)`
- `Collection(Edm.Double)`
- `Collection(Edm.Int32)`
- `Collection(Edm.Int64)`

Types such as `Edm.Int32` and `Edm.DateTimeOffset` support all six of the comparison operators: `eq`, `ne`, `lt`, `le`, `gt`, and `ge`. Lambda expressions over collections of these types can contain simple expressions using any of these operators. This applies to both `any` and `all`. For example, these filters are allowed:

- `ratings/any(r: r ne 5)`
- `dates/any(d: d gt 2017-08-24T00:00:00Z)`
- `not margins/all(m: m eq 3.5)`

However, there are limitations on how such comparison expressions can be combined into more complex expressions inside a lambda expression:

- Rules for `any`:
 - Simple inequality expressions can't be usefully combined with any other expressions.
For example, this expression is allowed:
 - `ratings/any(r: r ne 5)`but this expression isn't:
 - `ratings/any(r: r ne 5 and r gt 2)`and while this expression is allowed, it isn't useful because the conditions overlap:
 - `ratings/any(r: r ne 5 or r gt 7)`
 - Simple comparison expressions involving `eq`, `lt`, `le`, `gt`, or `ge` can be combined with `and` / `or`. For example:
 - `ratings/any(r: r gt 2 and r le 5)`
 - `ratings/any(r: r le 5 or r gt 7)`
 - Comparison expressions combined with `and` (conjunctions) can be further combined using `or`. This form is known in Boolean logic as "[Disjunctive Normal Form](#)" (DNF).
For example:
 - `ratings/any(r: (r gt 2 and r le 5) or (r gt 7 and r lt 10))`

- Rules for `all`:
 - Simple equality expressions can't be usefully combined with any other expressions. For example, this expression is allowed:
 - `ratings/all(r: r eq 5)`
 - but this expression isn't:
 - `ratings/all(r: r eq 5 or r le 2)`
 - and while this expression is allowed, it isn't useful because the conditions overlap:
 - `ratings/all(r: r eq 5 and r le 7)`
 - Simple comparison expressions involving `ne`, `lt`, `le`, `gt`, or `ge` can be combined with `and` / `or`. For example:
 - `ratings/all(r: r gt 2 and r le 5)`
 - `ratings/all(r: r le 5 or r gt 7)`
 - Comparison expressions combined with `or` (disjunctions) can be further combined using `and`. This form is known in Boolean logic as "[Conjunctive Normal Form](#)" (CNF). For example:
 - `ratings/all(r: (r le 2 or gt 5) and (r lt 7 or r ge 10))`

Rules for filtering complex collections

Lambda expressions over complex collections support a much more flexible syntax than lambda expressions over collections of primitive types. You can use any filter construct inside such a lambda expression that you can use outside one, with only two exceptions.

First, the functions `search.ismatch` and `search.ismatchscoring` aren't supported inside lambda expressions. For more information, see [Understanding OData collection filters in Azure AI Search](#).

Second, referencing fields that aren't *bound* to the range variable (so-called *free variables*) isn't allowed. For example, consider the following two equivalent OData filter expressions:

1. `stores/any(s: s/amenities/any(a: a eq 'parking')) and details/margin gt 0.5`
2. `stores/any(s: s/amenities/any(a: a eq 'parking' and details/margin gt 0.5))`

The first expression is allowed, while the second form is rejected because `details/margin` isn't bound to the range variable `s`.

This rule also extends to expressions that have variables bound in an outer scope. Such variables are free with respect to the scope in which they appear. For example, the first

expression is allowed, while the second equivalent expression isn't allowed because `s/name` is free with respect to the scope of the range variable `a`:

1. `stores/any(s: s/amenities/any(a: a eq 'parking') and s/name ne 'Flagship')`
2. `stores/any(s: s/amenities/any(a: a eq 'parking' and s/name ne 'Flagship'))`

This limitation shouldn't be a problem in practice since it's always possible to construct filters such that lambda expressions contain only bound variables.

Cheat sheet for collection filter rules

The following table summarizes the rules for constructing valid filters for each collection data type.

[Expand table](#)

Data type	Features allowed in lambda expressions with <code>any</code>	Features allowed in lambda expressions with <code>all</code>
<code>Collection(Edm.ComplexType)</code>	Everything except <code>search.ismatch</code> and <code>search.ismatchscoring</code>	Same
<code>Collection(Edm.String)</code>	Comparisons with <code>eq</code> or <code>search.in</code>	Comparisons with <code>ne</code> or <code>not search.in()</code>
	Combining sub-expressions with <code>or</code>	Combining sub-expressions with <code>and</code>
<code>Collection(Edm.Boolean)</code>	Comparisons with <code>eq</code> or <code>ne</code>	Same
<code>Collection(Edm.GeographyPoint)</code>	Using <code>geo.distance</code> with <code>lt</code> or <code>le</code> <code>geo.intersects</code>	Using <code>geo.distance</code> with <code>gt</code> or <code>ge</code> <code>not geo.intersects(...)</code>
	Combining sub-expressions with <code>or</code>	Combining sub-expressions with <code>and</code>
<code>Collection(Edm.DateTimeOffset),</code> <code>Collection(Edm.Double),</code> <code>Collection(Edm.Int32),</code> <code>Collection(Edm.Int64)</code>	Comparisons using <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>gt</code> , <code>le</code> , or <code>ge</code>	Comparisons using <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>gt</code> , <code>le</code> , or <code>ge</code>
	Combining comparisons with other sub-expressions using <code>or</code>	Combining comparisons with other sub-expressions using <code>and</code>

Data type	Features allowed in lambda expressions with <code>any</code>	Features allowed in lambda expressions with <code>all</code>
	<p>Combining comparisons except <code>ne</code> with other sub-expressions using <code>and</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Disjunctive Normal Form (DNF) ↴</p>	<p>Combining comparisons except <code>eq</code> with other sub-expressions using <code>or</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Conjunctive Normal Form (CNF) ↴</p>

For examples of how to construct valid filters for each case, see [How to write valid collection filters](#).

If you write filters often, and understanding the rules from first principles would help you more than just memorizing them, see [Understanding OData collection filters in Azure AI Search](#).

Next steps

- [Understanding OData collection filters in Azure AI Search](#)
- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Text normalization for case-insensitive filtering, faceting and sorting

09/28/2025

In Azure AI Search, a *normalizer* is a component that pre-processes text for keyword matching over fields marked as "filterable", "facetable", or "sortable". In contrast with full text "searchable" fields that are paired with [text analyzers](#), content that's created for filter-facet-sort operations doesn't undergo analysis or tokenization. Omission of text analysis can produce unexpected results when casing and character differences show up, which is why you need a normalizer to homogenize variations in your content.

By applying a normalizer, you can achieve light text transformations that improve results:

- Consistent casing (such as all lowercase or uppercase)
- Normalize accents and diacritics like ö or ê to ASCII equivalent characters "o" and "e"
- Map characters like - and whitespace into a user-specified character

Benefits of normalizers

Searching and retrieving documents from a search index requires matching the query input to the contents of the document. Matching is either over tokenized content, as is the case when you invoke "search", or over non-tokenized content if the request is a [filter](#), [facet](#), or [orderby](#) operation.

Because non-tokenized content is also not analyzed, small differences in the content are evaluated as distinctly different values. Consider the following examples:

- `$filter=City eq 'Las Vegas'` will only return documents that contain the exact text "Las Vegas" and exclude documents with "LAS VEGAS" and "las vegas", which is inadequate when the use-case requires all documents regardless of the casing.
- `search=*&facet=City,count:5` will return "Las Vegas", "LAS VEGAS" and "las vegas" as distinct values despite being the same city.
- `search=usa&$orderby=City` will return the cities in lexicographical order: "Las Vegas", "Seattle", "las vegas", even if the intent is to order the same cities together irrespective of the case.

A normalizer, which is invoked during indexing and query execution, adds light transformations that smooth out minor differences in text for filter, facet, and sort scenarios. In the previous

examples, the variants of "Las Vegas" would be processed according to the normalizer you select (for example, all text is lower-cased) for more uniform results.

How to specify a normalizer

Normalizers are specified in an index definition, on a per-field basis, on text fields (`Edm.String` and `Collection(Edm.String)`) that have at least one of "filterable", "sortable", or "facetable" properties set to true. Setting a normalizer is optional and is null by default. We recommend evaluating predefined normalizers before configuring a custom one.

Normalizers can only be specified when you add a new field to the index, so if possible, try to assess the normalization needs upfront and assign normalizers in the initial stages of development when dropping and recreating indexes is routine.

1. When creating a field definition in the [index](#), set the "normalizer" property to one of the following values: a [predefined normalizer](#) such as "lowercase", or a custom normalizer (defined in the same index schema).

JSON

```
"fields": [
{
  "name": "Description",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "filterable": true,
  "analyzer": "en.microsoft",
  "normalizer": "lowercase"
  ...
}
]
```

2. Custom normalizers are defined in the "normalizers" section of the index first, and then assigned to the field definition as shown in the previous step. For more information, see [Create Index](#) and also [Add custom normalizers](#).

JSON

```
"fields": [
{
  "name": "Description",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "analyzer": null,
}
```

```
    "normalizer": "my_custom_normalizer"  
},
```

!**Note**

To change the normalizer of an existing field, [rebuild the index](#) entirely (you cannot rebuild individual fields).

A good workaround for production indexes, where rebuilding indexes is costly, is to create a new field identical to the old one but with the new normalizer, and use it in place of the old one. Use [Update Index](#) to incorporate the new field and [mergeOrUpload](#) to populate it. Later, as part of planned index servicing, you can clean up the index to remove obsolete fields.

Predefined and custom normalizers

Azure AI Search provides built-in normalizers for common use-cases along with the capability to customize as required.

 [Expand table](#)

Category	Description
Predefined normalizers	Provided out-of-the-box and can be used without any configuration.
Custom normalizers ¹	For advanced scenarios. Requires user-defined configuration of a combination of existing elements, consisting of char and token filters.

⁽¹⁾ Custom normalizers don't specify tokenizers since normalizers always produce a single token.

Normalizers reference

Predefined normalizers

 [Expand table](#)

Name	Description and Options
standard	Lowercases the text followed by asciifolding.
lowercase	Transforms characters to lowercase.

Name	Description and Options
uppercase	Transforms characters to uppercase.
asciifolding	Transforms characters that aren't in the Basic Latin Unicode block to their ASCII equivalent, if one exists. For example, changing à to a.
elision	Removes elision from beginning of the tokens.

Supported char filters

Normalizers support two character filters that are identical to their counterparts in [custom analyzer character filters](#):

- [mapping ↗](#)
- [pattern_replace ↗](#)

Supported token filters

The list below shows the token filters supported for normalizers and is a subset of the overall [token filters used in custom analyzers](#).

- [arabic_normalization ↗](#)
- [asciifolding ↗](#)
- [cjk_width ↗](#)
- [elision ↗](#)
- [german_normalization ↗](#)
- [hindi_normalization ↗](#)
- [indic_normalization ↗](#)
- [persian_normalization ↗](#)
- [scandinavian_normalization ↗](#)
- [scandinavian_folding ↗](#)
- [sorani_normalization ↗](#)
- [lowercase ↗](#)
- [uppercase ↗](#)

Add custom normalizers

Custom normalizers are [defined within the index schema](#). The definition includes a name, a type, one or more character filters and token filters. The character filters and token filters are the building blocks for a custom normalizer and responsible for the processing of the text. These filters are applied from left to right.

The `token_filter_name_1` is the name of token filter, and `char_filter_name_1` and `char_filter_name_2` are the names of char filters (see [supported token filters](#) and [supported char filters](#) tables below for valid values).

JSON

```
"normalizers":(optional)[
  {
    "name": "name of normalizer",
    "@odata.type": "#Microsoft.Azure.Search.CustomNormalizer",
    "charFilters": [
      "char_filter_name_1",
      "char_filter_name_2"
    ],
    "tokenFilters": [
      "token_filter_name_1"
    ]
  }
],
"charFilters":(optional)[
  {
    "name": "char_filter_name_1",
    "@odata.type": "#char_filter_type",
    "option1": "value1",
    "option2": "value2",
    ...
  }
],
"tokenFilters":(optional)[
  {
    "name": "token_filter_name_1",
    "@odata.type": "#token_filter_type",
    "option1": "value1",
    "option2": "value2",
    ...
  }
]
```

Custom normalizers can be added during index creation or later by updating an existing one. Adding a custom normalizer to an existing index requires the "allowIndexDowntime" flag to be specified in [Update Index](#) and will cause the index to be unavailable for a few seconds.

Custom normalizer example

The example below illustrates a custom normalizer definition with corresponding character filters and token filters. Custom options for character filters and token filters are specified separately as named constructs, and then referenced in the normalizer definition as illustrated below.

- A custom normalizer named "my_custom_normalizer" is defined in the "normalizers" section of the index definition.
- The normalizer is composed of two character filters and three token filters: elision, lowercase, and customized asciifolding filter "my_asciifolding".
- The first character filter "map_dash" replaces all dashes with underscores while the second one "remove_whitespace" removes all spaces.

JSON

```
{
  "name": "myindex",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": false,
    },
    {
      "name": "city",
      "type": "Edm.String",
      "filterable": true,
      "facetable": true,
      "normalizer": "my_custom_normalizer"
    }
  ],
  "normalizers": [
    {
      "name": "my_custom_normalizer",
      "@odata.type": "#Microsoft.Azure.Search.CustomNormalizer",
      "charFilters": [
        "map_dash",
        "remove_whitespace"
      ],
      "tokenFilters": [
        "my_asciifolding",
        "elision",
        "lowercase",
      ]
    }
  ],
  "charFilters": [
    {
      "name": "map_dash",
      "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
      "mappings": [ "-=>_"]
    },
    {
      "name": "remove_whitespace",
      "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
      "mappings": [ "\u0020=>" ]
    }
  ]
}
```

```
        }
    ],
    "tokenFilters":[
        {
            "name":"my_asciifolding",
            "@odata.type": "#Microsoft.Azure.Search.AsciiFoldingTokenFilter",
            "preserveOriginal":true
        }
    ]
}
```

See also

- [Querying concepts in Azure AI Search](#)
- [Analyzers for linguistic and text processing](#)
- [Search Documents REST API](#)

Add faceted navigation to search results

Faceted navigation is used for self-directed filtering on query results in a search app, where your application offers form controls for scoping search to groups of documents (for example, categories or brands), and Azure AI Search provides the data structures and filters to back the experience.

In this article, learn the steps for returning a faceted navigation structure in Azure AI Search. Once you're familiar with basic concepts and clients, continue to [Facet examples](#) for syntax about various use cases, including basic facetting and distinct counts.

More facet capabilities are available through preview APIs:

- hierarchical facet structures
- facet filtering
- facet aggregations

[Facet navigation examples](#) provide the syntax and usage for the preview features.

Faceted navigation in a search page

Facets are dynamic because they're based on each specific query result set. A search response brings with it all of the facet buckets used to navigate the documents in the result. The query executes first, and then facets are pulled from the current results and assembled into a faceted navigation structure.

In Azure AI Search, facets are one layer deep and can't be hierarchical unless you use the preview API. If you aren't familiar with faceted navigation structures, the following example shows one on the left. Counts indicate the number of matches for each facet. The same document can be represented in multiple facets.



Hotels Search - Facet Navigation

wifi



Category:

Budget (5)
Luxury (5)
Resort and Spa (5)
Boutique (3)
Extended-Stay (1)

Amenities:

free wifi (11)
laundry service (6)
24-hour front desk service (5)
pool (5)
restaurant (5)
concierge (4)
continental breakfast (4)
free parking (4)
view (4)
coffee in lobby (3)
air conditioning (2)
bar (1)

19 Results

Super Deluxe Inn & Suites

Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor/Outdoor Pool, Fitness Center, Florida Green certified, Starbucks Coffee, HDTV

Category: Boutique

Amenities: bar, free wifi, free wifi

Double Sanctuary Resort

5* Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Conde Nast Traveler. Free WiFi, Flexible check in/out, Fitness Center & Nespresso in room.

Category: Resort and Spa

Amenities: view, laundry service, free wifi

Pull'r Inn Motel

The hotel rooms and suites offer the perfect blend of beauty and elegance. Our rooms will elevate your stay, whether you're traveling for business, celebrating a honeymoon, or just looking for a remarkable getaway. With views of the valley or the iconic fountains right from your suite, your stay will be nothing short of unforgettable.

|< < 1 2 3 4 5 > >|

Facets can help you find what you're looking for, while ensuring that you don't get zero results. As a developer, facets let you expose the most useful search criteria for navigating your search index.

Faceted navigation in code

Facets are enabled on supported fields in an index, and then specified on a query. The faceted navigation structure is returned at the beginning of the response, followed by the results.

The following REST example is an empty query (`"search": "*"`) that is scoped to the entire index (see the [built-in hotels sample](#)). The `facets` parameter specifies the "Category" field.

HTTP

```
POST https://{{service_name}}.search.windows.net/indexes/hotels/docs/search?api-version={{api_version}}
{
  "search": "*",
  "queryType": "simple",
  "select": "",
  "searchFields": "",
  "filter": "",
```

```
        "facets": [ "Category"],
        "orderby": "",
        "count": true
    }
```

The response for the example starts with the faceted navigation structure. The structure consists of "Category" values and a count of the hotels for each one. It's followed by the rest of the search results, trimmed here to just one document for brevity. This example works well for several reasons. The number of facets for this field fall under the limit (default is 10) so all of them appear, and every hotel in the index of 50 hotels is represented in exactly one of these categories.

JSON

```
{
    "@odata.context": "https://demo-search-
svc.search.windows.net/indexes('hotels')/$metadata#docs(*)",
    "@odata.count": 50,
    "@search.facets": {
        "Category": [
            {
                "count": 13,
                "value": "Budget"
            },
            {
                "count": 12,
                "value": "Resort and Spa"
            },
            {
                "count": 9,
                "value": "Luxury"
            },
            {
                "count": 7,
                "value": "Boutique"
            },
            {
                "count": 5,
                "value": "Suite"
            },
            {
                "count": 4,
                "value": "Extended-Stay"
            }
        ],
        "value": [
            {
                "@search.score": 1.0,
                "HotelId": "1",
                "HotelName": "Stay-Kay City Hotel",
                "Description": "The hotel is ideally located on the main commercial"
            }
        ]
    }
}
```

```
artery of the city in the heart of New York. A few minutes away is Time's Square and  
the historic centre of the city, as well as other places of interest that make New  
York one of America's most attractive and cosmopolitan cities.",  
    "Category": "Boutique",  
    "Tags": [  
        "pool",  
        "air conditioning",  
        "concierge"  
    ],  
    "ParkingIncluded": false,  
},  
...  
]  
}
```

Enable facets on fields

You can add facets to new fields that contain plain text or numeric content. Supported data types include strings, dates, boolean fields, and numeric fields (but not vectors).

You can use the Azure portal, REST APIs, Azure SDKs or any method that supports the creation or update of index schemas in Azure AI Search. As a first step, identify which fields to use for faceting.

Choose which fields to attribute

Facets can be calculated over single-value fields and collections. Fields that work best in faceted navigation have these characteristics:

- Human readable (nonvector) content.
- Low cardinality (a few distinct values that repeat throughout documents in your search corpus).
- Short descriptive values (one or two words) that render nicely in a navigation tree.

The values within a field, and not the field name itself, produce the facets in a faceted navigation structure. If the facet is a string field named *Color*, facets are blue, green, and any other value for that field. Review field values to ensure there are no typos, nulls, or casing differences. Consider [assigning a normalizer](#) to a filterable and facetable field to smooth out minor variations in the text. For example, "Canada", "CANADA", and "canada" would all be normalized to one bucket.

Avoid unsupported fields

You can't set facets on existing fields, on vector fields, or fields of type `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)`.

On complex field collections, "facetable" must be null.

Start with new field definitions

Attributes that affect how a field is indexed can only be set when fields are created. This restriction applies to facets and filters.

If your index already exists, you can add a new field definition that provides facets. Existing documents in the index get a null value for the new field. This null value is replaced the next time you [refresh the index](#).

Azure portal

1. In the search services page of the [Azure portal](#), go to the **Fields** tab of the index and select **Add field**.
2. Provide a name, data type, and attributes. We recommend adding filterable because it's common to set filters based on a facet bucket in the response. We recommend sortable because filters produce unordered results, and you might want to sort them in your application.

You can also set searchable if you also want to support full text search on the field, and retrievable if you want to include the field in the search response.

The screenshot shows the Microsoft Azure portal interface for managing an index named 'hotels'. At the top, there are navigation links for 'Home', 'Copilot', and a user profile. Below the header, the index details are shown: 'Documents: 12', 'Total storage: 61.15 KB', 'Vector index quota usage: 0 Bytes', and 'Max storage: 15 GB'. The 'Fields' tab is currently selected. In the 'Fields' section, there is a table listing fields like 'Id', 'Modified', 'IsDeleted', 'HotelName', 'Category', 'City', 'State', and 'Description', each with its type (String, DateTimeOffset, Int32) and various searchable attributes (Retrievable, Filterable, Sortable, Facetable). The 'Category' field is highlighted with a red box around its row, and its 'Facetable' checkbox is also checked. To the right of the table, there is a panel titled 'Index Field' with fields for 'Field name' (set to 'CategoryFacets') and 'Type' (set to 'Edm.String'). Below this is a 'Configure attributes' section with checkboxes for 'Retrievable', 'Filterable' (which is checked), 'Sortable' (which is checked), and 'Facetable' (which is checked). There are also 'Searchable' and 'Autocomplete settings' checkboxes. At the bottom right of the main area are 'Save' and 'Cancel' buttons.

3. Save the field definition.

Return facets in a query

Recall that facets are dynamically calculated from results in a query response. You only get facets for documents found by the current query.

Azure portal

Use JSON view in Search Explorer to set facet parameters in the [Azure portal ↗](#).

1. Select an index and open Search Explorer in JSON View.
2. Provide a query in JSON. You can type it out, copy the JSON from a REST example, or use intellisense to help with syntax. Refer to the REST example in the next tab for reference on facet expressions.
3. Select **Search** to return faceted results, articulated in JSON.

Here's a screenshot of the [basic facet query example](#) on the [hotels sample index](#). You can paste in other examples in this article to return the results in Search Explorer.

hotels-sample-index ... X

Save Discard Refresh Create demo app Edit JSON Delete Encryption

Documents Total storage Vector index quota usage Max storage

50 561.54 KB 0 Bytes 15 GB

Search explorer Fields CORS Scoring profiles Semantic configurations Vector profiles

2024-11-01-preview View

JSON query editor

```
2 "search": "*",
3 "facets": [
4   "Category",
5   "Tags,count:5",
6   "Rating,values:1|2|3|4|5"
7 ],
8 "count": true
```

Query view
Image view
JSON view

Results

```
3 "@odata.count": 50,
4 "@search.facets": {
5   "Rating": [...],
6   "Tags": [...],
7   "Category": [
8     {
9       "value": "Budget",
10      "count": 13
11    },
12    {
13      "value": "Suite",
14      "count": 12
15    },
16    {
17      "value": "Boutique",
18      "count": 7
19    },
20    {
21      "value": "Resort and Spa",
22      "count": 7
23    }
24  ]
25}
```

Search

Best practices for working with facets

This section is a collection of tips and workarounds that are helpful for application development.

We recommend the [C#: Add search to web apps](#) for an example of faceted navigation that includes code for the presentation layer. The sample also includes filters, suggestions, and autocomplete. It uses JavaScript and React for the presentation layer.

Initialize a faceted navigation structure with an unqualified or empty search string

It's useful to initialize a search page with an open query (`"search": "*"`) to completely fill in the faceted navigation structure. As soon as you pass query terms in the request, the faceted navigation structure is scoped to just the matches in the results, rather than the entire index. This practice is helpful for verifying facet and filter behaviors during testing. If you include match criteria in the query, the response excludes documents that don't match, which has the potential downstream effect of excluding facets.

Clear facets

When you design the user experience, remember to add a mechanism for clearing facets. A common approach for clearing facets is issuing an open query to reset the page.

Disable facetting to save on storage and improve performance

For performance and storage optimization, set `"facetable": false` for fields that should never be used as a facet. Examples include string fields for unique values, such as an ID or product name, to prevent their accidental (and ineffective) use in faceted navigation. This best practice is especially important for the REST API, which enables filters and facets on string fields by default.

Remember that you can't use `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)` fields in faceted navigation. Recall that facets work best on fields with low cardinality. Due to how geo-coordinates resolve, it's rare that any two sets of coordinates are equal in a given dataset. As such, facets aren't supported for geo-coordinates. You should use a city or region field to facet by location.

Check for bad data

As you prepare data for indexing, check fields for null values, misspellings or case discrepancies, and single and plural versions of the same word. By default, filters and facets don't undergo lexical analysis or [spell check](#), which means that all values of a "facetable" field are potential facets, even if the words differ by one character.

[Normalizers](#) can mitigate data discrepancies, correcting for casing and character differences. Otherwise, to inspect your data, you can check fields at their source, or run queries that return values from the index.

An index isn't the best place to fix nulls or invalid values. You should fix data problems in your source, assuming it's a database or persistent storage, or in a data cleansing step that you perform prior to indexing.

Ordering facet buckets

Although you can sort within a bucket, there's no parameters for controlling the order of facet buckets in the navigation structure as a whole. If you want facet buckets in a specific order, you must provide it in application code.

Discrepancies in facet counts

Under certain circumstances, you might find that facet counts aren't fully accurate due to the [sharding architecture](#). Every search index is spread across multiple shards, and each shard reports the top N facets by document count, which are then combined into a single result. Because it's just the top N facets for each shard, it's possible to miss or under-count matching documents in the facet response.

To guarantee accuracy, you can artificially inflate the count:<number> to a large number to force full reporting from each shard. You can specify `"count": "0"` for unlimited facets. Or, you can set "count" to a value that's greater than or equal to the number of unique values of the faceted field. For example, if you're faceting by a "size" field that has five unique values, you could set `"count:5"` to ensure all matches are represented in the facet response.

The tradeoff with this workaround is increased query latency, so use it only when necessary.

Preserve a facet navigation structure asynchronously of filtered results

In Azure AI Search, facets exist for current results only. However, it's a common application requirement to retain a static set of facets so that the user can navigate in reverse, retracing steps to explore alternative paths through search content.

If you want a static set of facets alongside a dynamic drilldown experience, you can implement it by using two filtered queries: one scoped to the results, the other used to create a static list of facets for navigation purposes.

Offset large facet counts through filters

Search results and facet results that are too large can be trimmed by [adding filters](#). In the following example, in the query for *cloud computing*, 254 items have *internal specification* as a content type. If results are too large, adding filters can help your users refine the query by adding more criteria.

Items aren't mutually exclusive. If an item meets the criteria of both filters, it's counted in each one. This duplication is possible when faceting on `Collection(Edm.String)` fields, which are often used to implement document tagging.

Output

Search term: "cloud computing"

Content type

Internal specification (254)

Video (10)

Next steps

[Facet navigation examples](#)

Last updated on 11/05/2025

Faceted navigation examples

This section extends [faceted navigation configuration](#) with examples that demonstrate basic usage and other scenarios.

Facetable fields are defined in an index, but facet parameters and expressions are defined in query requests. If you have an index with facetable fields, you can try new preview features like [facet hierarchies](#), [facet aggregations](#), and [facet filters](#) on existing indexes.

Facet parameters and syntax

Depending on the API, a facet query is usually an array of facet expressions that are applied to search results. Each facet expression contains a facetable field name, optionally followed by a comma-separated list of name-value pairs.

- *facet query* is a query request that includes a facet property.
- *facetable field* is a field definition in the search index attributed with the `facetable` property.
- *count* is the number of matches for each facet found in the search results.

The following table describes facet parameters used in the examples.

 [Expand table](#)

Facet parameter	Description	Usage	Example
<code>count</code>	Maximum number of facet terms per structure.	Integer. Default is 10. There's no upper limit, but higher values degrade performance, especially if the faceted field contains a large number of unique terms. This is due to the way facet queries are distributed across shards. You can set <code>count</code> to zero or to a value that's greater than or equal to the number of unique values in the	<code>Tags,count:5</code> limits the faceted navigation response to 5 facet buckets that containing the most facet counts, but they can be in any order.

Facet	Description	Usage	Example
parameter		facetable field to get an accurate count across all shards. The tradeoff is increased latency.	
sort	Determines order of facet buckets.	Valid values are <code>count</code> , <code>-count</code> , <code>value</code> , <code>-value</code> . Use <code>count</code> to list facets from greatest to smallest. Use <code>-count</code> to sort in ascending order (smallest to greatest). Use <code>value</code> to sort alphanumerically by facet value in ascending order. Use <code>-value</code> to sort descending by value.	"facet=Category, count:3, sort:count" gets the top three facet buckets in search results, listed in descending order by the number of matches in each Category. If the top three categories are Budget, Extended-Stay, and Luxury, and Budget has 5 hits, Extended-Stay has 6, and Luxury has 4, then the facet buckets are ordered as Extended-Stay, Budget, Luxury. Another example is "facet=Rating, sort:-value". It produces facets for all possible ratings, in descending order by value. If ratings are from 1 to 5, the facets are ordered 5, 4, 3, 2, 1, irrespective of how many documents match each rating.
values	Provides values for facet labels.	Set to pipe-delimited numeric or <code>Edm.DateTimeOffset</code> values specifying a dynamic set of facet entry values. The values must be listed in sequential, ascending order to get the expected results.	"facet=baseRate, values:10 20" produces three facet buckets: one for base rate 0 up to but not including 10, one for 10 up to but not including 20, and one for 20 and higher. A string "facet=lastRenovationDate, values:2024-02-01T00:00:00Z" produces two facet buckets: one for hotels renovated before February 2024, and one for hotels renovated February 1, 2024 or later.
interval	Provides an interval sequence for facets that can be grouped into intervals.	An integer interval greater than zero for numbers, or minute, hour, day, week, month, quarter, year for date time values.	"facet=baseRate, interval:100" produces facet buckets based on base rate ranges of size 100. If base rates are all between \$60 and \$600, there are facet buckets for 0-100, 100-200, 200-300, 300-400, 400-500, and 500-600. The string "facet=lastRenovationDate, interval:year" produces one facet bucket for each year a hotel was renovated.
timeoffset	Specifies the UTC	Set to <code>([+-]hh:mm, [+-]hhmm,</code> or	"facet=lastRenovationDate, interval:day, timeoffset:-01:00" uses the day boundary that starts at 01:00:00 UTC (midnight

Facet	Description	Usage	Example
parameter	time offset to account for in setting time boundaries.	[+-]hh). If used, the parameter must be combined with the interval option, and only when applied to a field of type Edm.DateTimeOffset.	in the target time zone).

`count` and `sort` can be combined in the same facet specification, but they can't be combined with `interval` or `values`.

`interval` and `values` can't be combined together.

Interval facets on date time are computed based on the UTC time if `timeoffset` isn't specified. For example, for `"facet=lastRenovationDate,interval:day"`, the day boundary starts at 00:00:00 UTC.

Basic facet example

The following facet queries work against the [hotels sample index](#). You can use [JSON view](#) in Search Explorer to paste in the JSON query. For help with getting started, see [Add faceted navigation to search results](#).

This first query retrieves facets for Categories, Ratings, Tags, and rooms with baseRate values in specific ranges. Notice the last facet is on a subfield of the Rooms collection. Facets count the parent document (Hotels) and not intermediate subdocuments (Rooms), so the response determines the number of *hotels* that have any rooms in each pricing category.

rest

```
POST /indexes/hotels-sample-index/docs/search?api-version={{api_version}}
{
  "search": "ocean view",
  "facets": [ "Category", "Rating", "Tags", "Rooms/BaseRate,values:80|150|220" ],
  "count": true
}
```

This second example uses a filter to narrow down the previous faceted query result after the user selects Rating 3 and category "Motel".

rest

```
POST /indexes/hotels-sample-index/docs/search?api-version={{api_version}}
{
```

```
"search": "water view",
"facets": [ "Tags", "Rooms/BaseRate,values:80|150|220" ],
"filter": "Rating eq 3 and Category eq 'Motel'",
"count": true
}
```

The third example sets an upper limit on unique terms returned in a query. The default is 10, but you can increase or decrease this value using the count parameter on the facet attribute. This example returns facets for city, limited to 5.

rest

```
POST /indexes/hotels-sample-index/docs/search?api-version={{api_version}}
{
  "search": "view",
  "facets": [ "Address/City,count:5" ],
  "count": true
}
```

This example shows three facets for "Category", "Tags", and "Rating", with a count override on "Tags" and a range override for "Rating", which is otherwise stored as a double in the index.

HTTP

```
POST https://{{service_name}}.search.windows.net/indexes/hotels/docs/search?api-
version={{api_version}}
{
  "search": "*",
  "facets": [
    "Category",
    "Tags,count:5",
    "Rating,values:1|2|3|4|5"
  ],
  "count": true
}
```

For each faceted navigation tree, there's a default limit of the top 10 facet instances found by the query. This default makes sense for navigation structures because it keeps the values list to a manageable size. You can override the default by assigning a value to "count". For example, "Tags,count:5" reduces the number of tags under the Tags section to the top five.

For Numeric and DateTime values only, you can explicitly set values on the facet field (for example, `facet=Rating,values:1|2|3|4|5`) to separate results into contiguous ranges (either ranges based on numeric values or time periods). Alternatively, you can add "interval", as in `facet=Rating,interval:1`.

Each range is built using 0 as a starting point, a value from the list as an endpoint, and then trimmed of the previous range to create discrete intervals.

Distinct values example

You can formulate a query that returns a distinct value count for each facetable field. This example formulates an empty or unqualified query ("search": "*") that matches on all documents, but by setting `top` to zero, you get just the counts, with no results.

For brevity, this query includes just two fields marked as `facetable` in the hotels sample index.

HTTP

```
POST https://{{service_name}}.search.windows.net/indexes/hotels/docs/search?api-version={{api_version}}
{
  "search": "*",
  "count": true,
  "top": 0,
  "facets": [
    "Category", "Address/StateProvince"
  ]
}
```

Results from this query are as follows:

JSON

```
{
  "@odata.count": 50,
  "@search.facets": {
    "Address/StateProvince": [
      {
        "count": 9,
        "value": "WA"
      },
      {
        "count": 6,
        "value": "CA"
      },
      {
        "count": 4,
        "value": "FL"
      },
      {
        "count": 3,
        "value": "NY"
      },
      {
        "count": 3,
        "value": "OR"
      },
      {
        "count": 3,
        "value": "TX"
      }
    ]
  }
}
```

```

},
{
  "count": 2,
  "value": "GA"
},
{
  "count": 2,
  "value": "MA"
},
{
  "count": 2,
  "value": "TN"
},
{
  "count": 1,
  "value": "AZ"
}
],
"Category": [
  {
    "count": 13,
    "value": "Budget"
  },
  {
    "count": 12,
    "value": "Suite"
  },
  {
    "count": 7,
    "value": "Boutique"
  },
  {
    "count": 7,
    "value": "Resort and Spa"
  },
  {
    "count": 6,
    "value": "Extended-Stay"
  },
  {
    "count": 5,
    "value": "Luxury"
  }
]
},
"value": []
}

```

Facet hierarchy example

① Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Using the [latest preview REST API](#) or the Azure portal, you can configure a facet hierarchy using the `>` and `;` operators.

[+] Expand table

Operator	Description
<code>></code>	Nesting (hierarchical) operator denotes a parent–child relationship.
<code>;</code>	Semicolon operator denotes multiple fields at the same nesting level, which are all children of the same parent. The parent must contain only one field. Both the parent and child fields must be <code>facetable</code> .

The order of operations in a facet expression that includes facet hierarchies are:

- The options operator (comma `,`) that separates facet parameters for the facet field, such as the comma in `Rooms/BaseRate,values`
- The parentheses, such as the ones enclosing `(Rooms/BaseRate,values:50 ; Rooms/Type)`.
- The nesting operator (angled bracket `>`)
- The append operator (semicolon `;`), demonstrated in a second example `"Tags>(Rooms/BaseRate,values:50 ; Rooms/Type)"` in this section, where two child facets are peers under the Tags parent.

Notice that parentheses are processed before nesting and append operations: `A > B ; C` would be different than `A > (B ; C)`.

There are several examples for facet hierarchies. The first example is a query that returns just a few documents, which is helpful for viewing a full response. Facets count the parent document (Hotels) and not intermediate subdocuments (Rooms), so the response determines the number of *hotels* that have any rooms in each facet bucket.

rest

```
POST /indexes/hotels-sample-index/docs/search?api-version=2025-11-01-Preview
{
  "search": "ocean",
  "facets": ["Address/StateProvince>Address/City", "Tags>Rooms/BaseRate,values:50"],
  "select": "HotelName, Description, Tags, Address/StateProvince, Address/City",
  "count": true
}
```

Results from this query are as follows. Both hotels have pools. For other tags, only one hotel provides the amenity.

JSON

```
{  
    "@odata.count": 2,  
    "@search.facets": {  
        "Tags": [  
            {  
                "value": "pool",  
                "count": 2,  
                "@search.facets": {  
                    "Rooms/BaseRate": [  
                        {  
                            "to": 50,  
                            "count": 0  
                        },  
                        {  
                            "from": 50,  
                            "count": 2  
                        }  
                    ]  
                }  
            },  
            {  
                "value": "air conditioning",  
                "count": 1,  
                "@search.facets": {  
                    "Rooms/BaseRate": [  
                        {  
                            "to": 50,  
                            "count": 0  
                        },  
                        {  
                            "from": 50,  
                            "count": 1  
                        }  
                    ]  
                }  
            },  
            {  
                "value": "bar",  
                "count": 1,  
                "@search.facets": {  
                    "Rooms/BaseRate": [  
                        {  
                            "to": 50,  
                            "count": 0  
                        },  
                        {  
                            "from": 50,  
                            "count": 1  
                        }  
                    ]  
                }  
            }  
        ]  
    }  
}
```

```
        }
    },
{
    "value": "restaurant",
    "count": 1,
    "@search.facets": {
        "Rooms/BaseRate": [
            {
                "to": 50,
                "count": 0
            },
            {
                "from": 50,
                "count": 1
            }
        ]
    }
},
{
    "value": "view",
    "count": 1,
    "@search.facets": {
        "Rooms/BaseRate": [
            {
                "to": 50,
                "count": 0
            },
            {
                "from": 50,
                "count": 1
            }
        ]
    }
}
],
"Address/StateProvince": [
{
    "value": "FL",
    "count": 1,
    "@search.facets": {
        "Address/City": [
            {
                "value": "Tampa",
                "count": 1
            }
        ]
    }
},
{
    "value": "HI",
    "count": 1,
    "@search.facets": {
        "Address/City": [
            {
                "value": "Honolulu",
                "count": 1
            }
        ]
    }
}
```

```

        }
    ]
}
],
},
"value": [
{
    "@search.score": 1.6076145,
    "HotelName": "Ocean Water Resort & Spa",
    "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near the pier, rooftop pool, waterfront dining & more.",
    "Tags": [
        "view",
        "pool",
        "restaurant"
    ],
    "Address": {
        "City": "Tampa",
        "StateProvince": "FL"
    }
},
{
    "@search.score": 1.0594962,
    "HotelName": "Windy Ocean Motel",
    "Description": "Oceanfront hotel overlooking the beach features rooms with a private balcony and 2 indoor and outdoor pools. Inspired by the natural beauty of the island, each room includes an original painting of local scenes by the owner. Rooms include a mini fridge, Keurig coffee maker, and flatscreen TV. Various shops and art entertainment are on the boardwalk, just steps away.",
    "Tags": [
        "pool",
        "air conditioning",
        "bar"
    ],
    "Address": {
        "City": "Honolulu",
        "StateProvince": "HI"
    }
}
]
}

```

This second example extends the previous one, demonstrating multiple top-level facets with multiple children. Notice the semicolon (;) operator separates each child.

rest

```

POST /indexes/hotels-sample-index/docs/search?api-version=2025-11-01-Preview
{
    "search": "+ocean",
    "facets": ["Address/StateProvince > Address/City", "Tags > (Rooms/BaseRate,values:50
; Rooms/Type)"],
    "select": "HotelName, Description, Tags, Address/StateProvince, Address/City",

```

```
"count": true  
}
```

A partial response, trimmed for brevity, shows Tags with child facets for the rooms base rate and type. In the hotels sample index, both hotels that match to `+ocean` have rooms in each type and a pool.

JSON

```
{  
  "@odata.count": 2,  
  "@search.facets": {  
    "Tags": [  
      {  
        "value": "pool",  
        "count": 2,  
        "@search.facets": {  
          "Rooms/BaseRate": [  
            {  
              "to": 50,  
              "count": 0  
            },  
            {  
              "from": 50,  
              "count": 2  
            }  
          ],  
          "Rooms/Type": [  
            {  
              "value": "Budget Room",  
              "count": 2  
            },  
            {  
              "value": "Deluxe Room",  
              "count": 2  
            },  
            {  
              "value": "Standard Room",  
              "count": 2  
            },  
            {  
              "value": "Suite",  
              "count": 2  
            }  
          ]  
        }  
      }  
    ]  
  }  
}  
...  
}
```

This last example shows precedence rules for parentheses that affects nesting levels. Suppose you want to return a facet hierarchy in this order.

```
Address/StateProvince
```

```
  Address/City
```

```
    Category
```

```
    Rating
```

To return this hierarchy, create a query where Category and Rating are siblings under Address/City.

JSON

```
{  
  "search": "beach",  
  "facets": [  
    "Address/StateProvince > (Address/City > (Category ; Rating))"  
  ],  
  "select": "HotelName, Description, Tags, Address/StateProvince, Address/City",  
  "count": true  
}
```

If you remove the innermost parentheses, Category and Rating are no longer siblings because the precedence rules mean that the `>` operator is evaluated before `;`.

JSON

```
{  
  "search": "beach",  
  "facets": [  
    "Address/StateProvince > (Address/City > Category ; Rating)"  
  ],  
  "select": "HotelName, Description, Tags, Address/StateProvince, Address/City",  
  "count": true  
}
```

The top-level parent is still Address/StateProvince, but now Address/City and Rating are on same level.

```
Address/StateProvince
```

```
  Rating
```

```
  Address/City
```

```
    Category
```

Facet filtering example

 Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Using the [latest preview REST API](#) or the Azure portal, you can configure facet filters.

Facet filtering enables you to constrain the facet values returned to those matching a specified regular expression. Two new parameters accept a regular expression that is applied to the facet field:

- `includeTermFilter` filters the facet values to those that match the regular expression
- `excludeTermFilter` filters the facet values to those that don't match the regular expression

If a facet string satisfies both conditions, the `excludeTermFilter` takes precedence because the set of bucket strings is first evaluated with `includeTermFilter` and then excluded with `excludeTermFilter`.

Only those facet values that match the regular expression are returned. You can combine these parameters with other facet options (for example, `count`, `sort`, and [hierarchical faceting](#)) on string fields.

Because the regular expression is nested within a JSON string value, you must escape both the double quote (`"`) and the backslash (`\`) characters. The regular expression itself is delimited by the forward slash (`/`). For more information about escape patterns, see [Regular expression search](#).

The following example shows how to escape special characters in your regular expression such as backslash, double quotes, or regular expression syntax characters.

JSON

```
{  
  "search": "*",
  "facets": [
    "name,includeTermFilter:/EscapeBackslash\\\\OrDoubleQuote\\\"OrRegexCharacter\\/(/)"
  ]
}
```

Here's an example of a facet filter that matches on Budget and Extended-Stay hotels, with Rating as a child of each hotel category.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2025-11-01-Preview
{
  "search": "*",
  "facets": [ "(Category,includeTermFilter:/(Budget|Extended-"
```

```

Stay())>Rating,values:1|2|3|4|5"],
  "select": "HotelName, Category, Rating",
  "count": true
}

```

The following example is an abbreviated response (hotel documents are omitted for brevity).

JSON

```
{
  "@odata.count": 50,
  "@search.facets": {
    "Category": [
      {
        "value": "Budget",
        "count": 13,
        "@search.facets": {
          "Rating": [
            {
              "to": 1,
              "count": 0
            },
            {
              "from": 1,
              "to": 2,
              "count": 0
            },
            {
              "from": 2,
              "to": 3,
              "count": 4
            },
            {
              "from": 3,
              "to": 4,
              "count": 5
            },
            {
              "from": 4,
              "to": 5,
              "count": 4
            },
            {
              "from": 5,
              "count": 0
            }
          ]
        }
      },
      {
        "value": "Extended-Stay",
        "count": 6,
        "@search.facets": {
          "Rating": [
            {

```

```
        "to": 1,
        "count": 0
    },
{
    "from": 1,
    "to": 2,
    "count": 0
},
{
    "from": 2,
    "to": 3,
    "count": 4
},
{
    "from": 3,
    "to": 4,
    "count": 1
},
{
    "from": 4,
    "to": 5,
    "count": 1
},
{
    "from": 5,
    "count": 0
}
]
}
],
{
    "value": [ ALL 50 HOTELS APPEAR HERE ]
}
```

Facet aggregation example

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Using the [latest preview REST API](#) or the Azure portal, you can aggregate facets.

Facet aggregations allow you to compute metrics from facet values. The aggregation capability works alongside the existing faceting options.

Aggregator	Description
Sum	Returns the total accumulated value from the field across all documents. Applies to numeric types only. Supported in earlier preview releases.
Min	Returns the minimum value from the field across all documents. Applies to numeric types only.
Max	Returns the maximum value from the field across all documents. Applies to numeric types only.
Avg	Returns the average value from the field across all documents. Applies to numeric types only.
Cardinality	Returns the approximate count of distinct values from the field across all documents using the HyperLogLog algorithm . You can request <code>cardinality</code> on facetable fields, including string and datetime fields (along with their corresponding collection forms).

Setting precision thresholds for cardinality aggregation

On a cardinality aggregation, you can set a `precisionThreshold` option as a demarcation between counts that are expected to be close to accurate, and counts that can be less accurate. The maximum value is 40,000. The default value is 3,000.

Faceting is performed in memory. Increasing `precisionThreshold` results in more memory consumption (the `precisionThreshold` value multiplied by 8 bytes).

Example: Sum facet aggregation

You can sum any facetable field of a numeric data type (except vectors and geographic coordinates).

Here's an example using the hotels-sample-index. The Rooms/SleepsCount field is facetable and numeric, so we choose this field to demonstrate sum. If we sum that field, we get the sleep count for the entire hotel. Recall that facets count the parent document (Hotels) and not intermediate subdocuments (Rooms), so the response sums the SleepsCount of all rooms for the entire hotel. In this query, we add a filter to sum the SleepsCount for just one hotel.

rest

```
POST /indexes/hotels-sample-index/docs/search?api-version=2025-11-01-Preview

{
  "search": "*",
  "filter": "HotelId eq '41'",
  "facets": [ "Rooms/SleepsCount", metric: sum"],
  "select": "HotelId, HotelName, Rooms/Type, Rooms/SleepsCount",
```

```
        "count": true
    }
```

A response for the query might look like the following example. Windy Ocean Model can accommodate a total of 40 guests.

JSON

```
{
    "@odata.count": 1,
    "@search.facets": {
        "Rooms/SleepsCount": [
            {
                "sum": 40.0
            }
        ]
    },
    "value": [
        {
            "@search.score": 1.0,
            "HotelId": "41",
            "HotelName": "Windy Ocean Motel",
            "Rooms": [
                {
                    "Type": "Suite",
                    "SleepsCount": 4
                },
                {
                    "Type": "Deluxe Room",
                    "SleepsCount": 2
                },
                {
                    "Type": "Budget Room",
                    "SleepsCount": 2
                },
                {
                    "Type": "Budget Room",
                    "SleepsCount": 2
                },
                {
                    "Type": "Suite",
                    "SleepsCount": 2
                },
                {
                    "Type": "Standard Room",
                    "SleepsCount": 2
                },
                {
                    "Type": "Deluxe Room",
                    "SleepsCount": 2
                },
                {
                    "Type": "Suite",
                    "SleepsCount": 2
                }
            ],
            "TotalSleeps": 40
        }
    ]
}
```

```
{
  "Type": "Suite",
  "SleepsCount": 4
},
{
  "Type": "Standard Room",
  "SleepsCount": 4
},
{
  "Type": "Standard Room",
  "SleepsCount": 2
},
{
  "Type": "Deluxe Room",
  "SleepsCount": 2
},
{
  "Type": "Suite",
  "SleepsCount": 2
},
{
  "Type": "Standard Room",
  "SleepsCount": 2
},
{
  "Type": "Deluxe Room",
  "SleepsCount": 2
},
{
  "Type": "Deluxe Room",
  "SleepsCount": 2
},
{
  "Type": "Standard Room",
  "SleepsCount": 2
}
]
}
]
```

Example: A composite of all aggregations

Here's an example using a hypothetical 'facets' index that shows the syntax for each aggregation. Notice that cardinality has an extra `precisionThreshold` option (default is 3,000) set to 40,000 in this example.

HTTP

```
POST https://search-service.search.windows.net/indexes/facets/docs/search?api-version=2025-11-01-Preview
Authorization: Bearer {{token}}
Content-Type: application/json
```

```
{
  "search": "*",
  "facets": [ // field names are named <something>Value in this example
    "cardinalityValue, metric: cardinality,precisionThreshold: 40000",
    "sumValue,metric: sum",
    "avgValue,metric: avg",
    "minValue,metric: min",
    "maxValue,metric: max"
  ]
}
```

A response for the query might look like the following example.

JSON

```
{
  "@search.facets": {
    "cardinalityValue": [
      {
        "cardinality": 24000 // Number of distinct values in "cardinalityValue" field
      }
    ],
    "sumValue": [
      {
        "sum": 1200000 // Sum of all values in "sumValue" field
      }
    ],
    "avgValue": [
      {
        "avg": 50 // Average of all values in "avgValue" field
      }
    ],
    "minValue": [
      {
        "min": 1 // Minimum value in "minValue" field
      }
    ],
    "maxValue": [
      {
        "max": 100 // Maximum value in "maxValue" field
      }
    ]
  }
}
```

Example: Specify a default to substitute for missing values

All metrics support specifying a default value when a document doesn't contain a value.

- For nonstring types (numeric, datetime, boolean), set the `default` parameter to a specific value: `"default: 42"`.
- For string types, set the `default` parameter to a string, delimited using the single apostrophe delimiter: `"default: 'mystringhere'"`.

You can add a default value to use if a document contains a null for that field: `"facets": [{"Rooms/SleepsCount, metric: sum, default:2"}]`. If a room has a null value for the Rooms/SleepsCount field, the default substitutes for the missing value.

Here's a request that illustrates the default specification for each field type.

HTTP

```
POST https://search-service.search.windows.net/indexes/facets/docs/search?api-version=2025-11-01-Preview
Authorization: Bearer {{token}}
Content-Type: application/json

{
  "search": "*",
  "facets": [ // field names are named <datatype>Value in this example
    "stringField, metric: cardinality, default: 'my string goes here'",
    "doubleField,metric: sum, default: 5.0",
    "intField,metric: sum, default: 5",
    "longField,metric: sum, default: 5"
  ]
}
```

For string fields, a default value is delimited using the single quote character. To escape the character, prefix it with the backslash `\"`. All characters are valid within the string delimiters. The terminating character can't be a backslash.

Example: Multiple metrics on the same field

If the underlying data supports the use case, you can specify multiple metrics on the same field.

HTTP

```
POST https://search-service.search.windows.net/indexes/facets/docs/search?api-version=2025-11-01-Preview
Authorization: Bearer {{token}}
Content-Type: application/json

{
  "search": "*",
  "facets": [
    "fieldA, metric: cardinality, precisionThreshold: 40000",
    "fieldA, metric: sum",
    "fieldA, metric: count"
  ]
}
```

```
        "fieldA, metric: avg",
        "fieldA, metric: min",
        "fieldA, metric: max"
    ]
}
```

A response for the query might look like the following example.

JSON

```
{
  "@search.facets": {
    "fieldA": [
      {
        "cardinality": 24000 // Number of distinct values in "fieldA" field
      },
      {
        "sum": 1200000 // Sum of all values in " fieldA " field
      },
      {
        "avg": 5 // Avg of all values in " fieldA " field
      },
      {
        "min": 0 // Min of all values in " fieldA " field
      },
      {
        "max": 1200 // Max of all values in " fieldA " field
      }
    ]
  }
}
```

Next steps

Revisit [facet navigation configuration](#) for tools and APIs, and review [best practices](#) for working with facets in code.

We recommend the [C#: Add search to web apps](#) for an example of faceted navigation that includes code for the presentation layer. The sample also includes filters, suggestions, and autocomplete. It uses JavaScript and React for the presentation layer.

Shape search results or modify search results composition in Azure AI Search

05/29/2025

This article explains search results composition and how to shape full text search results to fit your scenarios. Search results are returned in a query response. The shape of a response is determined by parameters in the query itself. These parameters include:

- Number of matches found in the index (`count`)
- Number of matches returned in the response (50 by default, configurable through `top`) or per page (`skip` and `top`)
- A search score for each result, used for ranking (`@search.score`)
- Fields included in search results (`select`)
- Sort logic (`orderby`)
- Highlighting of terms within a result, matching on either the whole or partial term in the body
- Optional elements from the semantic ranker (`answers` at the top, `captions` for each match)

Search results can include top-level fields, but most of the response consists of matching documents in an array.

Clients and APIs for defining the query response

You can use the following clients to configure a query response:

- [Search Explorer](#) in the Azure portal, using JSON view so that you can specify any supported parameter
- [Documents - POST \(REST APIs\)](#)
- [SearchClient.Search Method \(Azure SDK for .NET\)](#)
- [SearchClient.Search Method \(Azure SDK for Python\)](#)
- [SearchClient.Search Method \(Azure for JavaScript\)](#)
- [SearchClient.Search Method \(Azure for Java\)](#)

Result composition

Results are mostly tabular, composed of fields of either all `retrievable` fields, or limited to just those fields specified in the `select` parameter. Rows are the matching documents, typically ranked in order of relevance unless your query logic precludes relevance ranking.

You can choose which fields are in search results. While a search document might have a large number of fields, typically only a few are needed to represent each document in results. On a query request, append `select=<field list>` to specify which `retrievable` fields should appear in the response.

Pick fields that offer contrast and differentiation among documents, providing sufficient information to invite a clickthrough response on the part of the user. On an e-commerce site, it might be a product name, description, brand, color, size, price, and rating. For the built-in hotels-sample index, it might be the "select" fields in the following example:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
  "search": "sandy beaches",
  "select": "HotelId, HotelName, Description, Rating, Address/City",
  "count": true
}
```

Tips for unexpected results

Occasionally, query output isn't what you're expecting to see. For example, you might find that some results appear to be duplicates, or a result that *should* appear near the top is positioned lower in the results. When query outcomes are unexpected, you can try these query modifications to see if results improve:

- Change `searchMode=any` (default) to `searchMode=all` to require matches on all criteria instead of any of the criteria. This is especially true when boolean operators are included in the query.
- Experiment with different lexical analyzers or custom analyzers to see if it changes the query outcome. The default analyzer breaks up hyphenated words and reduces words to root forms, which usually improves the robustness of a query response. However, if you need to preserve hyphens, or if strings include special characters, you might need to configure custom analyzers to ensure the index contains tokens in the right format. For more information, see [Partial term search and patterns with special characters \(hyphens, wildcard, regex, patterns\)](#).

Counting matches

The `count` parameter returns the number of documents in the index that are considered a match for the query. To return the count, add `count=true` to the query request. There's no

maximum value imposed by the search service. Depending on your query and the content of your documents, the count could be as high as every document in the index.

Count is accurate when the index is stable. If the system is actively adding, updating, or deleting documents, the count is approximate, excluding any documents that aren't fully indexed.

Count won't be affected by routine maintenance or other workloads on the search service. However if you have multiple partitions and a single replica, you could experience short-term fluctuations in document count (several minutes) as the partitions are restarted.

Tip

To check indexing operations, you can confirm whether the index contains the expected number of documents by adding `count=true` on an empty search `search=*` query. The result is the full count of documents in your index.

When testing query syntax, `count=true` can quickly tell you whether your modifications are returning greater or fewer results, which can be useful feedback.

Number of results in the response

Azure AI Search uses server-side paging to prevent queries from retrieving too many documents at once. Query parameters that determine the number of results in a response are `top` and `skip`. `top` refers to the number of search results in a page. `skip` is an interval of `top`, and it tells the search engine how many results to skip before getting the next set.

The default page size is 50, while the maximum page size is 1,000. If you specify a value greater than 1,000 and there are more than 1,000 results found in your index, only the first 1,000 results are returned. If the number of matches exceed the page size, the response includes information to retrieve the next page of results. For example:

JSON

```
"@odata.nextLink": "https://contoso-search-eastus.search.windows.net/indexes/realestate-us-sample-index/docs/search?api-version=2024-07-01"
```

The top matches are determined by search score, assuming the query is full text search or semantic. Otherwise, the top matches are an arbitrary order for exact match queries (where uniform `@search.score=1.0` indicates arbitrary ranking).

Set `top` to override the default of 50. In newer preview APIs, if you're using a hybrid query, you can specify `maxTextRecallSize` to return up to 10,000 documents.

To control the paging of all documents returned in a result set, use `top` and `skip` together. This query returns the first set of 15 matching documents plus a count of total matches.

HTTP

```
POST https://contoso-search-eastus.search.windows.net/indexes/realestate-us-sample-index/docs/search?api-version=2024-07-01

{
  "search": "condos with a view",
  "count": true,
  "top": 15,
  "skip": 0
}
```

This query returns the second set, skipping the first 15 to get the next 15 (16 through 30):

HTTP

```
POST https://contoso-search-eastus.search.windows.net/indexes/realestate-us-sample-index/docs/search?api-version=2024-07-01

{
  "search": "condos with a view",
  "count": true,
  "top": 15,
  "skip": 15
}
```

The results of paginated queries aren't guaranteed to be stable if the underlying index is changing. Paging changes the value of `skip` for each page, but each query is independent and operates on the current view of the data as it exists in the index at query time (in other words, there's no caching or snapshot of results, such as those found in a general purpose database).

Following is an example of how you might get duplicates. Assume an index with four documents:

JSON

```
{ "id": "1", "rating": 5 }
{ "id": "2", "rating": 3 }
{ "id": "3", "rating": 2 }
{ "id": "4", "rating": 1 }
```

Now assume you want results returned two at a time, ordered by rating. You would execute this query to get the first page of results: `$top=2&$skip=0&$orderby=rating desc`, producing the following results:

JSON

```
{ "id": "1", "rating": 5 }
{ "id": "2", "rating": 3 }
```

On the service, assume a fifth document is added to the index in between query calls: `{ "id": "5", "rating": 4 }`. Shortly thereafter, you execute a query to fetch the second page: `$top=2&$skip=2&$orderby=rating desc`, and get these results:

JSON

```
{ "id": "2", "rating": 3 }
{ "id": "3", "rating": 2 }
```

Notice that document 2 is fetched twice. This is because the new document 5 has a greater value for rating, so it sorts before document 2 and lands on the first page. While this behavior might be unexpected, it's typical of how a search engine behaves.

Paging through a large number of results

An alternative technique for paging is to use a [sort order](#) and [range filter](#) as a workaround for `skip`.

In this workaround, sort and filter are applied to a document ID field or another field that is unique for each document. The unique field must have `filterable` and `sortable` attribution in the search index.

1. Issue a query to return a full page of sorted results.

HTTP

```
POST /indexes/good-books/docs/search?api-version=2024-07-01
{
  "search": "divine secrets",
  "top": 50,
  "orderby": "id asc"
}
```

2. Choose the last result returned by the search query. An example result with only an ID value is shown here.

JSON

```
{  
  "id": "50"  
}
```

3. Use that ID value in a range query to fetch the next page of results. This ID field should have unique values, otherwise pagination might include duplicate results.

HTTP

```
POST /indexes/good-books/docs/search?api-version=2024-07-01  
{  
  "search": "divine secrets",  
  "top": 50,  
  "orderby": "id asc",  
  "filter": "id ge 50"  
}
```

4. Pagination ends when the query returns zero results.

ⓘ Note

The `filterable` and `sortable` attributes can only be enabled when a field is first added to an index, they cannot be enabled on an existing field.

Ordering results

In a full text search query, results can be ranked by:

- a search score
- a semantic reranker score
- a sort order on a `sortable` field

You can also boost any matches found in specific fields by adding a scoring profile.

Order by search score

For full text search queries, results are automatically [ranked by a search score](#) using a BM25 algorithm, calculated based on term frequency, document length, and average document length.

The `@search.score` range is either unbounded, or 0 up to (but not including) 1.00 on older services.

For either algorithm, a `@search.score` equal to 1.00 indicates an unscored or unranked result set, where the 1.0 score is uniform across all results. Unscored results occur when the query form is fuzzy search, wildcard or regex queries, or an empty search (`search=*`). If you need to impose a ranking structure over unscored results, consider an `orderby` expression to achieve that objective.

Order by the semantic reranker

If you're using [semantic ranker](#), the `@search.rerankerScore` determines the sort order of your results.

The `@search.rerankerScore` range is 1 to 4.00, where a higher score indicates a stronger semantic match.

Order with orderby

If consistent ordering is an application requirement, you can define an [orderby expression](#) on a field. Only fields that are indexed as "sortable" can be used to order results.

Fields commonly used in an `orderby` include rating, date, and location. Filtering by location requires that the filter expression calls the [geo.distance\(\)](#) function, in addition to the field name.

Numeric fields (`Edm.Double`, `Edm.Int32`, `Edm.Int64`) are sorted in numeric order (for example, 1, 2, 10, 11, 20).

String fields (`Edm.String`, `Edm.ComplexType` subfields) are sorted in either [ASCII sort order](#) or [Unicode sort order](#), depending on the language.

- Numeric content in string fields is sorted alphabetically (1, 10, 11, 2, 20).
- Upper case strings are sorted ahead of lower case (APPLE, Apple, BANANA, Banana, apple, banana). You can assign a [text normalizer](#) to preprocess the text before sorting to change this behavior. Using the lowercase tokenizer on a field has no effect on sorting behavior because Azure AI Search sorts on a nonanalyzed copy of the field.
- Strings that lead with diacritics appear last (Äpfel, Öffnen, Üben)

Boost relevance using a scoring profile

Another approach that promotes order consistency is using a [custom scoring profile](#). Scoring profiles give you more control over the ranking of items in search results, with the ability to boost matches found in specific fields. The extra scoring logic can help override minor differences among replicas because the search scores for each document are farther apart. We recommend the [ranking algorithm](#) for this approach.

Hit highlighting

Hit highlighting refers to text formatting (such as bold or yellow highlights) applied to matching terms in a result, making it easy to spot the match. Highlighting is useful for longer content fields, such as a description field, where the match isn't immediately obvious.

Notice that highlighting is applied to individual terms. There's no highlight capability for the contents of an entire field. If you want to highlight over a phrase, you have to provide the matching terms (or phrase) in a quote-enclosed query string. This technique is described further on in this section.

Hit highlighting instructions are provided on the [query request](#). Queries that trigger query expansion in the engine, such as fuzzy and wildcard search, have limited support for hit highlighting.

Requirements for hit highlighting

- Fields must be `Edm.String` or `Collection(Edm.String)`
- Fields must be attributed at `searchable`

Specify highlighting in the request

To return highlighted terms, include the `highlight` parameter in the query request. The parameter is set to a comma-delimited list of fields.

By default, the format mark up is ``, but you can override the tag using `highlightPreTag` and `highlightPostTag` parameters. Your client code handles the response (for example, applying a bold font or a yellow background).

HTTP

```
POST /indexes/good-books/docs/search?api-version=2024-07-01
{
    "search": "divine secrets",
    "highlight": "title, original_title",
    "highlightPreTag": "<b>",
```

```
        "highlightPostTag": "</b>"  
    }  
}
```

By default, Azure AI Search returns up to five highlights per field. You can adjust this number by appending a dash followed by an integer. For example, `"highlight": "description-10"` returns up to 10 highlighted terms on matching content in the description field.

Highlighted results

When highlighting is added to the query, the response includes an `@search.highlights` for each result so that your application code can target that structure. The list of fields specified for "highlight" are included in the response.

In a keyword search, each term is scanned for independently. A query for "divine secrets" returns matches on any document containing either term.

The screenshot shows the Azure AI Search UI for the "good-books" index. The search bar contains the query `search=divine secrets&$select=original_title&highlight=original_title`. The results table displays three documents with their scores and original titles. The first document is "Divine Secrets of the Ya-Ya Sisterhood" with a score of 19.593246. The second is "Divine Madness" with a score of 12.779835. The third is "Grave Secrets" with a score of 12.62534. In the JSON results, the `@search.highlights` field is shown for each document, where the matched terms ("Divine", "Secrets", "Ya-Ya Sisterhood", "Madness", "Grave") are wrapped in `` tags. The screenshot also shows the API version as 2020-06-30-Preview and the Request URL as `https://westus2.search.windows.net/indexes/good-books/docs?api-version=2020-06-30-Preview&search=divine%20secrets&%24select=original_title&highlight=original_title`.

Rank	Title	Score
1	Divine Secrets of the Ya-Ya Sisterhood	19.593246
2	Divine Madness	12.779835
3	Grave Secrets	12.62534

```
4  {  
5      "@search.score": 19.593246,  
6      "@search.highlights": {  
7          "original_title": [  
8              "<em>Divine</em> <em>Secrets</em> of the Ya-Ya Sisterhood"  
9          ]  
10     },  
11     "original_title": "Divine Secrets of the Ya-Ya Sisterhood"  
12   },  
13   {  
14       "@search.score": 12.779835,  
15       "@search.highlights": {  
16           "original_title": [  
17               "<em>Divine</em> Madness"  
18           ]  
19       },  
20       "original_title": "Divine Madness"  
21     },  
22     {  
23         "@search.score": 12.62534,  
24         "@search.highlights": {  
25             "original_title": [  
26                 "Grave <em>Secrets</em>"  
27             ]  
28         },
```

Keyword search highlighting

Within a highlighted field, formatting is applied to whole terms. For example, on a match against "The Divine Secrets of the Ya-Ya Sisterhood", formatting is applied to each term

separately, even though they're consecutive.

JSON

```
"@odata.count": 39,
"value": [
  {
    "@search.score": 19.593246,
    "@search.highlights": {
      "original_title": [
        "<em>Divine</em> <em>Secrets</em> of the Ya-Ya Sisterhood"
      ],
      "title": [
        "<em>Divine</em> <em>Secrets</em> of the Ya-Ya Sisterhood"
      ]
    },
    "original_title": "Divine Secrets of the Ya-Ya Sisterhood",
    "title": "Divine Secrets of the Ya-Ya Sisterhood"
  },
  {
    "@search.score": 12.779835,
    "@search.highlights": {
      "original_title": [
        "<em>Divine</em> Madness"
      ],
      "title": [
        "<em>Divine</em> Madness (Cherub, #5)"
      ]
    },
    "original_title": "Divine Madness",
    "title": "Divine Madness (Cherub, #5)"
  },
  {
    "@search.score": 12.62534,
    "@search.highlights": {
      "original_title": [
        "Grave <em>Secrets</em>"
      ],
      "title": [
        "Grave <em>Secrets</em> (Temperance Brennan, #5)"
      ]
    },
    "original_title": "Grave Secrets",
    "title": "Grave Secrets (Temperance Brennan, #5)"
  }
]
```

Phrase search highlighting

Whole-term formatting applies even on a phrase search, where multiple terms are enclosed in double quotation marks. The following example is the same query, except that "divine secrets"

is submitted as a quotation-enclosed phrase (some REST clients require that you escape the interior quotation marks with a backslash \"):

HTTP

```
POST /indexes/good-books/docs/search?api-version=2024-07-01
{
    "search": "\"divine secrets\"",
    "select": "title,original_title",
    "highlight": "title",
    "highlightPreTag": "<b>",
    "highlightPostTag": "</b>",
    "count": true
}
```

Because the criteria now have both terms, only one match is found in the search index. The response to the previous query looks like this:

JSON

```
{
    "@odata.count": 1,
    "value": [
        {
            "@search.score": 19.593246,
            "@search.highlights": {
                "title": [
                    "<b>Divine</b> <b>Secrets</b> of the Ya-Ya Sisterhood"
                ],
                "original_title": "Divine Secrets of the Ya-Ya Sisterhood",
                "title": "Divine Secrets of the Ya-Ya Sisterhood"
            }
        }
    ]
}
```

Phrase highlighting on older services

Search services that were created before July 15, 2020 implement a different highlighting experience for phrase queries.

For the following examples, assume a query string that includes the quote-enclosed phrase "super bowl". Before July 2020, any term in the phrase is highlighted:

JSON

```
"@search.highlights": {
    "sentence": [
```

```
"The <em>super</em> <em>bowl</em> is <em>super</em> awesome with a  
<em>bowl</em> of chips"
```

```
]
```

For search services created after July 2020, only phrases that match the full phrase query are returned in `@search.highlights`:

JSON

```
"@search.highlights": {  
    "sentence": [  
        "The <em>super</em> <em>bowl</em> is super awesome with a bowl of chips"  
    ]
```

Next steps

To quickly generate a search page for your client, consider these options:

- [Create demo app](#), in the Azure portal, creates an HTML page with a search bar, faceted navigation, and a thumbnail area if you have images.
- [Add search to an ASP.NET Core \(MVC\) app](#) is a tutorial and code sample that builds a functional client.
- [Add search to web apps](#) is a C# tutorial and code sample that uses the React JavaScript libraries for the user experience. The app is deployed using Azure Static Web Apps and it implements pagination.

Return a semantic answer in Azure AI Search

When invoking [semantic ranking and captions](#), you can optionally extract content from the top-matching documents that "answers" the query directly. One or more answers can be included in the response, which you can then render on a search page to improve the user experience of your app.

A semantic answer is verbatim content in your search index that a reading comprehension model has recognized as an answer to the query posed in the request. It's not a generated answer. For guidance on a chat-style user interaction model that uses generative AI to compose answers from your content, see [Retrieval Augmented Generation \(RAG\)](#).

In this article, learn how to request a semantic answer, unpack the response, and find out what content characteristics are most conducive to producing high-quality answers.

Prerequisites

All prerequisites that apply to [semantic queries](#) also apply to answers, including [service tier and region](#).

- Query logic must include the semantic query parameters "queryType=semantic", plus the "answers" parameter. Required parameters are discussed in this article.
- Query strings entered by the user must be recognizable as a question (what, where, when, how).
- Search documents in the index must contain text having the characteristics of an answer, and that text must exist in one of the fields listed in the [semantic configuration](#). For example, given a query "what is a hash table", if none of the fields in the semantic configuration contain passages that include "A hash table is ...", then it's unlikely an answer is returned.

What is a semantic answer?

A semantic answer is a substructure of a [semantic query response](#). It consists of one or more verbatim passages from a search document, formulated as an answer to a query that looks like a question. To return an answer, phrases or sentences must exist in a search document that have the language characteristics of an answer, and the query itself must be posed as a question.

Azure AI Search uses a machine reading comprehension model to recognize and pick the best answer. The model produces a set of potential answers from the available content, and when it reaches a high enough confidence level, it proposes one as an answer.

Answers are returned as an independent, top-level object in the query response payload that you can choose to render on search pages, along side search results. Structurally, it's an array element within the response consisting of text, a document key, and a confidence score.

Formulate a REST query for "answers"

To return a semantic answer, the query must have the semantic `"queryType"`, `"queryLanguage"`, `"semanticConfiguration"`, and the `"answers"` parameters. Specifying these parameters doesn't guarantee an answer, but the request must include them for answer processing to occur.

JSON

```
{  
  "search": "how do clouds form",  
  "queryType": "semantic",  
  "queryLanguage": "en-us",  
  "semanticConfiguration": "my-semantic-config",  
  "answers": "extractive|count-3",  
  "captions": "extractive|highlight-true",  
  "count": "true"  
}
```

- A query string must not be null and should be formulated as question.
- `"queryType"` must be set to "semantic".
- `"queryLanguage"` must be one of the values from the [supported languages list \(REST API\)](#).
- A `"semanticConfiguration"` determines which string fields provide tokens to the extraction model. The same fields that produce captions also produce answers. See [Create a semantic configuration](#) for details.
- For `"answers"`, parameter construction is `"answers": "extractive"`, where the default number of answers returned is one. You can increase the number of answers by adding a `count` as shown in the above example, up to a maximum of 10. Whether you need more than one answer depends on the user experience of your app, and how you want to render results.

Unpack an "answer" from the response

Answers are provided in the `"@search.answers"` array, which appears first in the query response. Each answer in the array includes:

- Document key
- Text or content of the answer, in plain text or with formatting
- Confidence score

If an answer is indeterminate, the response shows up as `"@search.answers": []`. The answers array is followed by the value array, which is the standard response in a semantic query.

Given the query "how do clouds form" which can be directed at an index built on [content from the NASA Earth Book](#), the following example illustrates a verbatim answer (found on page 38):

JSON

```
{  
  "@search.answers": [  
    {  
      "key": "4123",  
      "text": "Sunlight heats the land all day, warming that moist air and causing it to rise high into the atmosphere until it cools and condenses into water droplets. Clouds generally form where air is ascending (over land in this case), but not where it is descending (over the river).",  
      "highlights": "Sunlight heats the land all day, warming that moist air and causing it to rise high into the atmosphere until it cools and condenses into water droplets. Clouds generally form<em> where air is ascending</em> (over land in this case), but not where it is<em> descending</em> (over the river).",  
      "score": 0.94639826  
    },  
    ],  
    "value": [  
      {  
        "@search.score": 0.5479723,  
        "@search.rerankerScore": 1.0321671911515296,  
        "@search.captions": [  
          {  
            "text": "Like all clouds, it forms when the air reaches its dew point—the temperature at which an air mass is cool enough for its water vapor to condense into liquid droplets. This false-color image shows valley fog, which is common in the Pacific Northwest of North America.",  
            "highlights": "Like all<em> clouds</em>, it<em> forms</em> when the air reaches its dew point—the temperature at which an air mass is cool enough for its water vapor to condense into liquid droplets. This false-color image shows valley<em> fog</em>, which is common in the Pacific Northwest of North America."  
          }  
        ],  
        "title": "Earth Atmosphere",  
        "content": "Fog is essentially a cloud lying on the ground. Like all clouds, it forms when the air reaches its dew point—the temperature at \\n\\nwhich an air mass is cool enough for its water vapor to condense into liquid
```

```
droplets.\n\nThis false-color image shows valley fog, which is common in the Pacific Northwest of North America. On clear winter nights, the \n\nground and overlying air cool off rapidly, especially at high elevations. Cold air is denser than warm air, and it sinks down into the \n\nvalleys. The moist air in the valleys gets chilled to its dew point, and fog forms. If undisturbed by winds, such fog may persist for \n\ndays. The Terra satellite captured this image of foggy valleys northeast of Vancouver in February 2010.\n\n",
```

```
    "locations": [
        "Pacific Northwest",
        "North America",
        "Vancouver"
    ]
}
]
```

When designing a search results page that includes answers, be sure to handle cases where answers aren't found.

Within @search.answers:

- "key" is the document key or ID of the match. Given a document key, you can use [Lookup Document API](#) to retrieve any or all parts of the search document to include on the search page or a detail page.
- "text" and "highlights" provide identical content, in both plain text and with highlights.

By default, highlights are styled as ``, which you can override using the existing `highlightPreTag` and `highlightPostTag` parameters. As noted elsewhere, the substance of an answer is verbatim content from a search document. The extraction model looks for characteristics of an answer to find the appropriate content, but doesn't compose new language in the response.

- "score" is a confidence score that reflects the strength of the answer. If there are multiple answers in the response, this score is used to determine the order. Top answers and top captions can be derived from different search documents, where the top answer originates from one document, and the top caption from another, but in general the same documents appear in the top positions within each array.

Answers are followed by the "value" array, which always includes scores, captions, and any fields that are retrievable by default. If you specified the select parameter, the "value" array is limited to the fields that you specified. See [Configure semantic ranker](#) for details.

Tips for producing high-quality answers

For best results, return semantic answers on a document corpus having the following characteristics:

- The "semanticConfiguration" must include fields that offer sufficient text in which an answer is likely to be found. Fields more likely to contain answers should be listed first in "prioritizedContentFields". Only verbatim text from a document can appear as an answer.
- Query strings must not be null (search=*) and the string should have the characteristics of a question, such as "what is" or "how to", as opposed to a keyword search consisting of terms or phrases in arbitrary order. If the query string doesn't appear to be a question, answer processing is skipped, even if the request specifies "answers" as a query parameter.
- Semantic extraction and summarization have limits over how many tokens per document can be analyzed in a timely fashion. In practical terms, if you have large documents that run into hundreds of pages, try to break up the content into smaller documents first.

Next steps

- [Semantic ranking overview](#)
- [Configure BM25 ranking](#)
- [Configure semantic ranker](#)

Last updated on 11/06/2025

Examples of *full* Lucene search syntax (advanced queries)

Article • 04/14/2025

When constructing queries for Azure AI Search, you can replace the default [simple query parser](#) with the more powerful [Lucene query parser](#) to formulate specialized and advanced query expressions.

The Lucene parser supports complex query formats, such as field-scoped queries, fuzzy search, infix and suffix wildcard search, proximity search, term boosting, and regular expression search. The extra power comes with more processing requirements so you should expect a slightly longer execution time. In this article, you can step through examples that demonstrate query operations based on full syntax.

! Note

Many of the specialized query constructions enabled through the full Lucene query syntax are not [text-analyzed](#), which can be surprising if you expect stemming or lemmatization. Lexical analysis is only performed on complete terms (a term query or phrase query). Query types with incomplete terms (prefix query, wildcard query, regex query, fuzzy query) are added directly to the query tree, bypassing the analysis stage. The only transformation performed on partial query terms is lowercasing.

Hotels sample index

The following queries are based on the hotels-sample-index, which you can create by following the instructions in this [quickstart](#).

Example queries are articulated using the REST API and POST requests. You can paste and run them in a [REST client](#). Or, use the JSON view of [Search Explorer](#) in the Azure portal. In JSON view, you can paste in the query examples shown here in this article.

Request headers must have the following values:

[] [Expand table](#)

Key	Value
Content-Type	application/json
api-key	<your-search-service-api-key>, either query or admin key

URI parameters must include your search service endpoint with the index name, docs collections, search command, and API version, similar to the following example:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2024-07-01
```

The request body should be formed as valid JSON:

JSON

```
{  
    "search": "*",  
    "queryType": "full",  
    "select": "HotelId, HotelName, Category, Tags, Description",  
    "count": true  
}
```

- `search` set to `*` is an unspecified query, equivalent to null or empty search. It's not especially useful, but it's the simplest search you can do, and it shows all retrievable fields in the index, with all values.
- `queryType` set to `full` invokes the full Lucene query parser and it's required for this syntax.
- `select` set to a comma-delimited list of fields is used for search result composition, including only those fields that are useful in the context of search results.
- `count` returns the number of documents matching the search criteria. On an empty search string, the count is all documents in the index (50 in the `hotels-sample-index`).

Example 1: Fielded search

Fielded search scopes individual, embedded search expressions to a specific field. This example searches for hotel names with the term `hotel` in them, but not `motel`. You can specify multiple fields using `AND`.

When you use this query syntax, you can omit the `searchFields` parameter when the fields you want to query are in the search expression itself. If you include `searchFields` with fielded search, the `fieldName:searchExpression` always takes precedence over `searchFields`.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
    "search": "HotelName:(hotel NOT motel) AND Category:'Boutique'", 
    "queryType": "full",
    "select": "HotelName, Category",
    "count": true
}
```

The response for this query should look similar to the following example, filtered on *Boutique*, returning hotels that include *hotel* in the name, while excluding results that include *motel* in the name.

JSON

```
{
    "@odata.count": 5,
    "value": [
        {
            "@search.score": 2.2289815,
            "HotelName": "Stay-Kay City Hotel",
            "Category": "Boutique"
        },
        {
            "@search.score": 1.3862944,
            "HotelName": "City Skyline Antiquity Hotel",
            "Category": "Boutique"
        },
        {
            "@search.score": 1.355046,
            "HotelName": "Old Century Hotel",
            "Category": "Boutique"
        },
        {
            "@search.score": 1.355046,
            "HotelName": "Sublime Palace Hotel",
            "Category": "Boutique"
        },
        {
            "@search.score": 1.355046,
            "HotelName": "Red Tide Hotel",
            "Category": "Boutique"
        }
    ]
}
```

The search expression can be a single term or a phrase, or a more complex expression in parentheses, optionally with Boolean operators. Some examples include the following:

- `HotelName:(hotel NOT motel)`

- Address/StateProvince:("WA" OR "CA")
- Tags:("free wifi" NOT "free parking") AND "coffee in lobby"

Be sure to put a phrase within quotation marks if you want both strings to be evaluated as a single entity, as in this case searching for two distinct locations in the Address/StateProvince field. Depending on the client, you might need to escape (\) the quotation marks.

The field specified in `fieldName:searchExpression` must be a searchable field. To learn how field definitions are attributed, see [Create Index \(REST API\)](#).

Example 2: Fuzzy search

Fuzzy search matches on terms that are similar, including misspelled words. To do a fuzzy search, append the tilde ~ symbol at the end of a single word with an optional parameter, a value between 0 and 2, that specifies the edit distance. For example, `blue~` or `blue~1` would return blue, blues, and glue.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
  "search": "Tags:conserge~",
  "queryType": "full",
  "select": "HotelName, Category, Tags",
  "searchFields": "HotelName, Category, Tags",
  "count": true
}
```

The response for this query resolves to `concierge` in the matching documents, trimmed for brevity:

JSON

```
{
  "@odata.count": 9,
  "value": [
    {
      "@search.score": 1.4947624,
      "HotelName": "Twin Vortex Hotel",
      "Category": "Luxury",
      "Tags": [
        "bar",
        "restaurant",
        "concierge"
      ]
    },
    {
```

```
    "@search.score": 1.1685618,  
    "HotelName": "Stay-Kay City Hotel",  
    "Category": "Boutique",  
    "Tags": [  
        "view",  
        "air conditioning",  
        "concierge"  
    ]  
,  
{  
    "@search.score": 1.1465473,  
    "HotelName": "Old Century Hotel",  
    "Category": "Boutique",  
    "Tags": [  
        "pool",  
        "free wifi",  
        "concierge"  
    ]  
,  
...  
}
```

Phrases aren't supported directly but you can specify a fuzzy match on each term of a multi-part phrase, such as `search=Tags:landy~ AND sevic~`. This query expression finds 15 matches on *laundry service*.

ⓘ Note

Fuzzy queries are not analyzed. Query types with incomplete terms (prefix query, wildcard query, regex query, fuzzy query) are added directly to the query tree, bypassing the analysis stage. The only transformation performed on partial query terms is lower casing.

Example 3: Proximity search

Proximity search finds terms that are near each other in a document. Insert a tilde `~` symbol at the end of a phrase followed by the number of words that create the proximity boundary.

This query searches for the terms *hotel* and *airport* within five words of each other in a document. The quotation marks are escaped (`\\"`) to preserve the phrase:

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01  
{  
    "search": "Description: \"hotel airport\"~5",  
    "queryType": "full",
```

```
    "select": "HotelName, Description",
    "searchFields": "HotelName, Description",
    "count": true
}
```

The response for this query should look similar to the following example:

JSON

```
{
  "@odata.count": 1,
  "value": [
    {
      "@search.score": 0.69167054,
      "HotelName": "Trails End Motel",
      "Description": "Only 8 miles from Downtown. On-site bar/restaurant, Free hot breakfast buffet, Free wireless internet, All non-smoking hotel. Only 15 miles from airport."
    }
  ]
}
```

Example 4: Term boosting

Term boosting refers to ranking a document higher if it contains the boosted term, relative to documents that don't contain the term. To boost a term, use the caret, `^`, symbol with a boost factor (a number) at the end of the term you're searching. The boost factor default is 1, and although it must be positive, it can be less than 1 (for example, 0.2). Term boosting differs from scoring profiles in that scoring profiles boost certain fields, rather than specific terms.

In this *before* query, search for *beach access* and notice that there are six documents that match on one or both terms.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
  "search": "beach access",
  "queryType": "full",
  "select": "HotelName, Description, Tags",
  "searchFields": "HotelName, Description, Tags",
  "count": true
}
```

In fact, only two documents match on *access*. The first instance is in second position, even though the document is missing the term *beach*.

JSON

```
{
  "@odata.count": 6,
  "value": [
    {
      "@search.score": 1.068669,
      "HotelName": "Johnson's Family Resort",
      "Description": "Family oriented resort located in the heart of the northland. Operated since 1962 by the Smith family, we have grown into one of the largest family resorts in the state. The home of excellent Smallmouth Bass fishing with 10 small cabins, we're a home not only to fishermen but their families as well. Rebuilt in the early 2000's, all of our cabins have all the comforts of home. Sporting a huge **beach** with multiple water toys for those sunny summer days and a Lodge full of games for when you just can't swim anymore, there's always something for the family to do. A full marina offers watercraft rentals, boat launch, powered dock slips, canoes (free to use), & fish cleaning facility. Rent pontoons, 14' fishing boats, 16' fishing rigs or jet ski's for a fun day or week on the water. Pets are accepted in the lakeside cottages.",
      "Tags": [
        "24-hour front desk service",
        "pool",
        "coffee in lobby"
      ]
    },
    {
      "@search.score": 1.0162708,
      "HotelName": "Campus Commander Hotel",
      "Description": "Easy **access** to campus and steps away from the best shopping corridor in the city. From meetings in town or gameday, enjoy our prime location between the union and proximity to the university stadium.",
      "Tags": [
        "free parking",
        "coffee in lobby",
        "24-hour front desk service"
      ]
    },
    {
      "@search.score": 0.9050383,
      "HotelName": "Lakeside B & B",
      "Description": "Nature is Home on the **beach**. Explore the shore by day, and then come home to our shared living space to relax around a stone fireplace, sip something warm, and explore the library by night. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackouts may apply.",
      "Tags": [
        "laundry service",
        "concierge",
        "free parking"
      ]
    },
    {
      "@search.score": 0.8955848,
      "HotelName": "Windy Ocean Motel",
      "Description": "Oceanfront hotel overlooking the **beach** features rooms"
    }
  ]
}
```

```

with a private balcony and 2 indoor and outdoor pools. Inspired by the natural
beauty of the island, each room includes an original painting of local scenes by
the owner. Rooms include a mini fridge, Keurig coffee maker, and flatscreen TV.
Various shops and art entertainment are on the boardwalk, just steps away.",

    "Tags": [
        "pool",
        "air conditioning",
        "bar"
    ],
},
{
    "@search.score": 0.83636594,
    "HotelName": "Happy Lake Resort & Restaurant",
    "Description": "The largest year-round resort in the area offering more of
everything for your vacation - at the best value! What can you enjoy while at the
resort, aside from the mile-long sandy **beaches** of the lake? Check out our
activities sure to excite both young and young-at-heart guests. We have it all,
including being named “Property of the Year” and a “Top Ten Resort” by top
publications.",
    "Tags": [
        "pool",
        "bar",
        "restaurant"
    ],
},
{
    "@search.score": 0.7808502,
    "HotelName": "Swirling Currents Hotel",
    "Description": "Spacious rooms, glamorous suites and residences, rooftop
pool, walking **access** to shopping, dining, entertainment and the city center.
Each room comes equipped with a microwave, a coffee maker and a minifridge. In-
room entertainment includes complimentary W-Fi and flat-screen TVs. ",
    "Tags": [
        "air conditioning",
        "laundry service",
        "24-hour front desk service"
    ],
}
]
}

```

In the *after* query, repeat the search, this time boosting results with the term *beach* over the term *access*. A human readable version of the query is `search=Description:beach^2 access`. Depending on your client, you might need to express `^2` as `%5E2`.

HTTP

```

POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
    "search": "Description:beach^2 access",
    "queryType": "full",
    "select": "HotelName, Description, Tags",

```

```
    "searchFields": "HotelName, Description, Tags",
    "count": true
}
```

After you boost the term *beach*, the match on Campus Commander Hotel moves down to fifth place.

JSON

```
{
  "@odata.count": 6,
  "value": [
    {
      "@search.score": 2.137338,
      "HotelName": "Johnson's Family Resort",
      "Description": "Family oriented resort located in the heart of the northland. Operated since 1962 by the Smith family, we have grown into one of the largest family resorts in the state. The home of excellent Smallmouth Bass fishing with 10 small cabins, we're a home not only to fishermen but their families as well. Rebuilt in the early 2000's, all of our cabins have all the comforts of home. Sporting a huge beach with multiple water toys for those sunny summer days and a Lodge full of games for when you just can't swim anymore, there's always something for the family to do. A full marina offers watercraft rentals, boat launch, powered dock slips, canoes (free to use), & fish cleaning facility. Rent pontoons, 14' fishing boats, 16' fishing rigs or jet ski's for a fun day or week on the water. Pets are accepted in the lakeside cottages.",
      "Tags": [
        "24-hour front desk service",
        "pool",
        "coffee in lobby"
      ]
    },
    {
      "@search.score": 1.8100766,
      "HotelName": "Lakeside B & B",
      "Description": "Nature is Home on the beach. Explore the shore by day, and then come home to our shared living space to relax around a stone fireplace, sip something warm, and explore the library by night. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackouts may apply.",
      "Tags": [
        "laundry service",
        "concierge",
        "free parking"
      ]
    },
    {
      "@search.score": 1.7911696,
      "HotelName": "Windy Ocean Motel",
      "Description": "Oceanfront hotel overlooking the beach features rooms with a private balcony and 2 indoor and outdoor pools. Inspired by the natural beauty of the island, each room includes an original painting of local scenes by the owner. Rooms include a mini fridge, Keurig coffee maker, and flatscreen TV. Various shops and art entertainment are on the boardwalk, just steps away."
    }
  ]
}
```

```

    "Tags": [
      "pool",
      "air conditioning",
      "bar"
    ],
  },
  {
    "@search.score": 1.6727319,
    "HotelName": "Happy Lake Resort & Restaurant",
    "Description": "The largest year-round resort in the area offering more of everything for your vacation - at the best value! What can you enjoy while at the resort, aside from the mile-long sandy beaches of the lake? Check out our activities sure to excite both young and young-at-heart guests. We have it all, including being named “Property of the Year” and a “Top Ten Resort” by top publications.",
    "Tags": [
      "pool",
      "bar",
      "restaurant"
    ],
  },
  {
    "@search.score": 1.0162708,
    "HotelName": "Campus Commander Hotel",
    "Description": "Easy access to campus and steps away from the best shopping corridor in the city. From meetings in town or gameday, enjoy our prime location between the union and proximity to the university stadium.",
    "Tags": [
      "free parking",
      "coffee in lobby",
      "24-hour front desk service"
    ],
  },
  {
    "@search.score": 0.7808502,
    "HotelName": "Swirling Currents Hotel",
    "Description": "Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center. Each room comes equipped with a microwave, a coffee maker and a minifridge. In-room entertainment includes complimentary W-Fi and flat-screen TVs. ",
    "Tags": [
      "air conditioning",
      "laundry service",
      "24-hour front desk service"
    ],
  }
]
}

```

Example 5: Regex

A regular expression search finds a match based on the contents between forward slashes / and lower-case strings, as documented in the [RegExp class](#).

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
  "search": "HotelName:/(Mo|Ho)tel/",
  "queryType": "full",
  "select": "HotelName",
  "count": true
}
```

The response for this query should look similar to the following example (trimmed for brevity):

JSON

```
{
  "@odata.count": 25,
  "value": [
    {
      "@search.score": 1,
      "HotelName": "Country Residence Hotel"
    },
    {
      "@search.score": 1,
      "HotelName": "Downtown Mix Hotel"
    },
    {
      "@search.score": 1,
      "HotelName": "Gastronomic Landscape Hotel"
    },
    . . .
    {
      "@search.score": 1,
      "HotelName": "Trails End Motel"
    },
    {
      "@search.score": 1,
      "HotelName": "Nordick's Valley Motel"
    },
    {
      "@search.score": 1,
      "HotelName": "King's Cellar Hotel"
    }
  ]
}
```

! Note

Regex queries are not analyzed. The only transformation performed on partial query terms is lower casing.

Example 6: Wildcard search

You can use generally recognized syntax for multiple (*) or single (?) character wildcard searches. The Lucene query parser supports the use of these symbols with a single term, and not a phrase.

In this query, search for hotel names that contain the prefix *sc*. You can't use a * or ? symbol as the first character of a search.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2024-07-01
{
  "search": "HotelName:sc*",
  "queryType": "full",
  "select": "HotelName",
  "count": true
}
```

The response for this query should look similar to the following example:

JSON

```
{
  "@odata.count": 1,
  "value": [
    {
      "@search.score": 1,
      "HotelName": "Waterfront Scottish Inn"
    }
  ]
}
```

(!) Note

Wildcard queries are not analyzed. The only transformation performed on partial query terms is lower casing.

Related content

Try specifying queries in code. The following link covers how to set up search queries using the Azure SDKs.

- [Quickstart: Full text search using the Azure SDKs](#)

More syntax reference, query architecture, and examples can be found in the following articles:

- [Full text search in Azure AI Search](#)
- [Simple query syntax](#)
- [Lucene query syntax in Azure AI Search](#)
- [OData \\$filter syntax in Azure AI Search](#)

Partial term search and patterns with special characters (hyphens, wildcard, regex, patterns)

Article • 04/14/2025

A *partial term search* refers to queries consisting of term fragments, where instead of a whole term, you might have just the beginning, middle, or end of term (sometimes referred to as prefix, infix, or suffix queries). A partial term search might include a combination of fragments, often with special characters such as hyphens, dashes, or slashes that are part of the query string. Common use-cases include parts of a phone number, URL, codes, or hyphenated compound words.

Partial terms and special characters can be problematic if the index doesn't have a token representing the text fragment you want to search for. During the [lexical analysis phase](#) of keyword indexing (assuming the default standard analyzer), special characters are discarded, compound words are split up, and whitespace is deleted. If you're searching for a text fragment that was modified during lexical analysis, the query fails because no match is found. Consider this example: a phone number like `+1 (425) 703-6214` (tokenized as `"1"`, `"425"`, `"703"`, `"6214"`) won't show up in a `"3-62"` query because that content doesn't actually exist in the index.

The solution is to invoke an analyzer during indexing that preserves a complete string, including spaces and special characters if necessary, so that you can include the spaces and characters in your query string. Having a whole, untokenized string enables pattern matching for "starts with" or "ends with" queries, where the pattern you provide can be evaluated against a term that isn't transformed by lexical analysis.

If you need to support search scenarios that call for analyzed and non-analyzed content, consider creating two fields in your index, one for each scenario. One field undergoes lexical analysis. The second field stores an intact string, using a content-preserving analyzer that emits whole-string tokens for pattern matching.

About partial term search

Azure AI Search scans for whole tokenized terms in the index and won't find a match on a partial term unless you include wildcard placeholder operators (`*` and `?`), or format the query as a regular expression.

Partial terms are specified using these techniques:

- [Regular expression queries](#) can be any regular expression that is valid under Apache Lucene.
- [Wildcard operators with prefix matching](#) refers to a generally recognized pattern that includes the beginning of a term, followed by `*` or `?` suffix operators, such as `search=cap*` matching on "Cap'n Jack's Waterfront Inn" or "Highline Capital". Prefixing matching is supported in both simple and full Lucene query syntax.
- [Wildcard with infix and suffix matching](#) places the `*` and `?` operators inside or at the beginning of a term, and requires regular expression syntax (where the expression is enclosed with forward slashes). For example, the query string (`search=/.*numeric.*/`) returns results on "alphanumeric" and "alphanumeric" as suffix and infix matches.

For regular expression, wildcard, and fuzzy search, analyzers aren't used at query time. For these query forms, which the parser detects by the presence of operators and delimiters, the query string is passed to the engine without lexical analysis. For these query forms, the analyzer specified on the field is ignored.

Note

When a partial query string includes characters, such as slashes in a URL fragment, you might need to add escape characters. In JSON, a forward slash `/` is escaped with a backward slash `\`. As such, `search=/.*microsoft.com\ azure\ ./` is the syntax for the URL fragment "microsoft.com/azure/".

Solving partial/pattern search problems

When you need to search on fragments or patterns or special characters, you can override the default analyzer with a custom analyzer that operates under simpler tokenization rules, retaining the entire string in the index.

The approach looks like this:

1. Define a second field to store an intact version of the string (assuming you want analyzed and non-analyzed text at query time)
2. Evaluate and choose among the various analyzers that emit tokens at the right level of granularity
3. Assign the analyzer to the field
4. Build and test the index

1 - Create a dedicated field

Analyzers determine how terms are tokenized in an index. Since analyzers are assigned on a per-field basis, you can create fields in your index to optimize for different scenarios. For example, you might define "featureCode" and "featureCodeRegex" to support regular full text search on the first, and advanced pattern matching on the second. The analyzers assigned to each field determine how the contents of each field are tokenized in the index.

JSON

```
{  
  "name": "featureCode",  
  "type": "Edm.String",  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": null  
},  
{  
  "name": "featureCodeRegex",  
  "type": "Edm.String",  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": "my_custom_analyzer"  
},
```

2 - Set an analyzer

When choosing an analyzer that produces whole-term tokens, the following analyzers are common choices:

 Expand table

Analyzer	Behaviors
language analyzers	Preserves hyphens in compound words or strings, vowel mutations, and verb forms. If query patterns include dashes, using a language analyzer might be sufficient.
keyword ↗	Content of the entire field is tokenized as a single term.
whitespace ↗	Separates on white spaces only. Terms that include dashes or other characters are treated as a single token.
custom analyzer	(recommended) Creating a custom analyzer lets you specify both the tokenizer and token filter. The previous analyzers must be used as-is. A custom analyzer lets you pick which tokenizers and token filters to use.

A recommended combination is the [keyword tokenizer ↗](#) with a [lower-case token](#)

Analyzer	Behaviors
	filter ↗ . By itself, the built-in keyword analyzer ↗ doesn't lower-case any upper-case text, which can cause queries to fail. A custom analyzer gives you a mechanism for adding the lower-case token filter.

Using a REST client, you can add the [Test Analyzer REST call](#) to inspect tokenized output.

The index must exist on the search service, but it can be empty. Given an existing index and a field containing dashes or partial terms, you can try various analyzers over specific terms to see what tokens are emitted.

1. First, check the Standard analyzer to see how terms are tokenized by default.

JSON

```
{
  "text": "SVP10-NOR-00",
  "analyzer": "standard"
}
```

2. Evaluate the response to see how the text is tokenized within the index. Notice how each term is lower-cased, hyphens removed, and substrings broken up into individual tokens. Only those queries that match on these tokens will return this document in the results. A query that includes "10-NOR" will fail.

JSON

```
{
  "tokens": [
    {
      "token": "svp10",
      "startOffset": 0,
      "endOffset": 5,
      "position": 0
    },
    {
      "token": "nor",
      "startOffset": 6,
      "endOffset": 9,
      "position": 1
    },
    {
      "token": "00",
      "startOffset": 10,
      "endOffset": 12,
      "position": 2
    }
  ]
}
```

```
    ]  
}
```

3. Now modify the request to use the `whitespace` or `keyword` analyzer:

JSON

```
{  
  "text": "SVP10-NOR-00",  
  "analyzer": "keyword"  
}
```

4. This time, the response consists of a single token, upper-cased, with dashes preserved as a part of the string. If you need to search on a pattern or a partial term such as "10-NOR", the query engine now has the basis for finding a match.

JSON

```
{  
  
  "tokens": [  
    {  
      "token": "SVP10-NOR-00",  
      "startOffset": 0,  
      "endOffset": 12,  
      "position": 0  
    }  
  ]  
}
```

ⓘ Important

Be aware that query parsers often lower-case terms in a search expression when building the query tree. If you are using an analyzer that does not lower-case text inputs during indexing, and you are not getting expected results, this could be the reason. The solution is to add a lower-case token filter, as described in the "Use custom analyzers" section below.

3 - Configure an analyzer

Whether you're evaluating analyzers or moving forward with a specific configuration, you'll need to specify the analyzer on the field definition, and possibly configure the analyzer itself if

you aren't using a built-in analyzer. When swapping analyzers, you typically need to rebuild the index (drop, recreate, and reload).

Use built-in analyzers

Built-in analyzers can be specified by name on an `analyzer` property of a field definition, with no extra configuration required in the index. The following example demonstrates how you would set the `whitespace` analyzer on a field.

For other scenarios and to learn more about other built-in analyzers, see [Built-in analyzers](#).

JSON

```
{  
  "name": "phoneNumber",  
  "type": "Edm.String",  
  "key": false,  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": "whitespace"  
}
```

Use custom analyzers

If you're using a [custom analyzer](#), define it in the index with a user-defined combination of tokenizer, token filter, with possible configuration settings. Next, reference it on a field definition, just as you would a built-in analyzer.

When the objective is whole-term tokenization, a custom analyzer that consists of a **keyword tokenizer** and **lower-case token filter** is recommended.

- The keyword tokenizer creates a single token for the entire contents of a field.
- The lowercase token filter transforms upper-case letters into lower-case text. Query parsers typically lowercase any uppercase text inputs. Lower-casing homogenizes the inputs with the tokenized terms.

The following example illustrates a custom analyzer that provides the keyword tokenizer and a lowercase token filter.

JSON

```
{  
  "fields": [  
    {  
      "name": "accountNumber",  
      "analyzer": "lowercase_keyword",  
      "tokenizer": "keyword",  
      "filter": "lowercase"  
    }  
  ]  
}
```

```

    "analyzer": "myCustomAnalyzer",
    "type": "Edm.String",
    "searchable": true,
    "filterable": true,
    "retrievable": true,
    "sortable": false,
    "facetable": false
  }
],
"analyzers": [
{
  "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
  "name": "myCustomAnalyzer",
  "charFilters": [],
  "tokenizer": "keyword_v2",
  "tokenFilters": ["lowercase"]
}
],
"tokenizers": [],
"charFilters": [],
"tokenFilters": []
}

```

➊ Note

The `keyword_v2` tokenizer and `lowercase` token filter are known to the system and using their default configurations, which is why you can reference them by name without having to define them first.

4 - Build and test

Once you've defined an index with analyzers and field definitions that support your scenario, load documents that have representative strings so that you can test partial string queries.

Use a REST client to query partial terms and special characters described in this article.

The previous sections explained the logic. This section steps through each API you should call when testing your solution.

- [Delete Index](#) removes an existing index of the same name so that you can recreate it.
- [Create Index](#) creates the index structure on your search service, including analyzer definitions and fields with an analyzer specification.
- [Load Documents](#) imports documents having the same structure as your index, as well as searchable content. After this step, your index is ready to query or test.

- [Test Analyzer](#) was introduced in [Set an analyzer](#). Test some of the strings in your index using various analyzers to understand how terms are tokenized.
- [Search Documents](#) explains how to construct a query request, using either [simple syntax](#) or [full Lucene syntax](#) for wildcard and regular expressions.

For partial term queries, such as querying "3-6214" to find a match on "+1 (425) 703-6214", you can use the simple syntax: `search=3-6214&queryType=simple`.

For infix and suffix queries, such as querying "num" or "numeric" to find a match on "alphanumeric", use the full Lucene syntax and a regular expression:

`search=/.*num.*/&queryType=full`

Optimizing prefix and suffix queries

Matching prefixes and suffixes using the default analyzer requires additional query features. Prefixes require [wildcard search](#) and suffixes require [regular expression search](#). Both of these features can reduce query performance.

The following example adds an [EdgeNGramTokenFilter](#) to make prefix or suffix matches faster. Tokens are generated in 2-25 character combinations that include characters. Here's an example progression from two to seven tokens: MS, MSF, MSFT, MSFT/, MSFT/S, MSFT/SQ, MSFT/SQ. `EdgeNGramTokenFilter` requires a `side` parameter which determines which side of the string character combinations are generated from. Use `front` for prefix queries and `back` for suffix queries.

Extra tokenization results in a larger index. If you have sufficient capacity to accommodate the larger index, this approach with its faster response time might be the best solution.

JSON

```
{
  "fields": [
    {
      "name": "accountNumber_prefix",
      "indexAnalyzer": "ngram_front_analyzer",
      "searchAnalyzer": "keyword",
      "type": "Edm.String",
      "searchable": true,
      "filterable": false,
      "retrievable": true,
      "sortable": false,
      "facetable": false
    },
    {
      "name": "accountNumber_suffix",
      "indexAnalyzer": "ngram_back_analyzer",
      "searchAnalyzer": "keyword",
      "type": "Edm.String",
      "searchable": true,
      "filterable": false,
      "retrievable": true,
      "sortable": false,
      "facetable": false
    }
  ]
}
```

```

    "indexAnalyzer": "ngram_back_analyzer",
    "searchAnalyzer": "keyword",
    "type": "Edm.String",
    "searchable": true,
    "filterable": false,
    "retrievable": true,
    "sortable": false,
    "facetable": false
  }
],
"analyzers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "ngram_front_analyzer",
    "charFilters": [],
    "tokenizer": "keyword_v2",
    "tokenFilters": ["lowercase", "front_edgeNGram"]
  },
  {
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "ngram_back_analyzer",
    "charFilters": [],
    "tokenizer": "keyword_v2",
    "tokenFilters": ["lowercase", "back_edgeNGram"]
  }
],
"tokenizers": [],
"charFilters": [],
"tokenFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.EdgeNGramTokenFilterV2",
    "name": "front_edgeNGram",
    "minGram": 2,
    "maxGram": 25,
    "side": "front"
  },
  {
    "@odata.type": "#Microsoft.Azure.Search.EdgeNGramTokenFilterV2",
    "name": "back_edgeNGram",
    "minGram": 2,
    "maxGram": 25,
    "side": "back"
  }
]
}

```

To search for account numbers that start with 123, we can use the following query:

```
{
  "search": "123",
```

```
"searchFields": "accountNumber_prefix"  
}
```

To search for account numbers that end with 456, we can use the following query:

```
{  
  "search": "456",  
  "searchFields": "accountNumber_suffix"  
}
```

Next steps

This article explains how analyzers both contribute to query problems and solve query problems. As a next step, take a closer look at analyzers affect indexing and query processing.

- [Tutorial: Create a custom analyzer for phone numbers](#)
- [Language analyzers](#)
- [Analyzers for text processing in Azure AI Search](#)
- [Analyze API \(REST\)](#)
- [How full text search works \(query architecture\)](#)

Fuzzy search to correct misspellings and typos

Article • 04/14/2025

Azure AI Search supports fuzzy search, a type of query that compensates for typos and misspelled terms in the input string. Fuzzy search scans for terms having a similar composition. Expanding search to cover near-matches has the effect of autocorrecting a typo when the discrepancy is just a few misplaced characters.

What is fuzzy search?

It's a query expansion exercise that produces a match on terms having a similar composition. When a fuzzy search is specified, the search engine builds a graph (based on [deterministic finite automaton theory](#)) of similarly composed terms, for all whole terms in the query. For example, if your query includes three terms "university of washington", a graph is created for every term in the query `search=university~ of~ washington~` (there's no stop-word removal in fuzzy search, so "of" gets a graph).

The graph consists of up to 50 expansions, or permutations, of each term, capturing both correct and incorrect variants in the process. The engine then returns the topmost relevant matches in the response.

For a term like "university", the graph might have "unversty, universty, university, universe, inverse". Any documents that match on those in the graph are included in results. In contrast with other queries that analyze the text to handle different forms of the same word ("mice" and "mouse"), the comparisons in a fuzzy query are taken at face value without any linguistic analysis on the text. "Universe" and "inverse", which are semantically different, will match because the syntactic discrepancies are small.

A match succeeds if the discrepancies are limited to two or fewer edits, where an edit is an inserted, deleted, substituted, or transposed character. The string correction algorithm that specifies the differential is the [Damerau-Levenshtein distance](#) metric. It's described as the "minimum number of operations (insertions, deletions, substitutions, or transpositions of two adjacent characters) required to change one word into the other".

In Azure AI Search:

- Fuzzy query applies to whole terms. Phrases aren't supported directly but you can specify a fuzzy match on each term of a multi-part phrase through AND constructions. For example, `search=dr~ AND cleanin~`. This query expression finds matches on "dry cleaning".

- The default distance of an edit is 2. A value of `~0` signifies no expansion (only the exact term is considered a match), but you could specify `~1` for one degree of difference, or one edit.
- A fuzzy query can expand a term up to 50 permutations. This limit isn't configurable, but you can effectively reduce the number of expansions by decreasing the edit distance to 1.
- Responses consist of documents containing a relevant match (up to 50).

During query processing, fuzzy queries don't undergo [lexical analysis](#). The query input is added directly to the query tree and expanded to create a graph of terms. The only transformation performed is lower casing.

Collectively, the graphs are submitted as match criteria against tokens in the index. As you can imagine, fuzzy search is inherently slower than other query forms. The size and complexity of your index can determine whether the benefits are enough to offset the latency of the response.

Note

Because fuzzy search tends to be slow, it might be worthwhile to investigate alternatives such as n-gram indexing, with its progression of short character sequences (two and three character sequences for bigram and trigram tokens). Depending on your language and query surface, n-gram might give you better performance. The trade off is that n-gram indexing is very storage intensive and generates much bigger indexes.

Another alternative, which you could consider if you want to handle just the most egregious cases, would be a [synonym map](#). For example, mapping "search" to "serach, serch, sarch", or "retrieve" to "retreive".

Indexing for fuzzy search

String fields that are attributed as "searchable" are candidates for fuzzy search.

Analyzers aren't used to create an expansion graph, but that doesn't mean analyzers should be ignored in fuzzy search scenarios. Analyzers are important for tokenization during indexing, where tokens in the inverted indexes are used for matching against the graph.

As always, if test queries aren't producing the matches you expect, experiment with different indexing analyzers. For example, try a [language analyzer](#) to see if you get better results. Some languages, particularly those with vowel mutations, can benefit from the inflection and irregular word forms generated by the Microsoft natural language processors. In some cases,

using the right language analyzer can make a difference in whether a term is tokenized in a way that is compatible with the value provided by the user.

How to invoke fuzzy search

Fuzzy queries are constructed using the full Lucene query syntax, invoking the [full Lucene query parser](#), and appending a tilde character ~ after each whole term entered by the user.

Here's an example of a query request that invokes fuzzy search. It includes four terms, two of which are misspelled:

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2024-07-01
{
    "search": "seattle~ waterfront~ view~ hotle~",
    "queryType": "full",
    "searchMode": "any",
    "searchFields": "HotelName, Description",
    "select": "HotelName, Description, Address/City",
    "count": "true"
}
```

1. Set the query type to the full Lucene syntax (`queryType=full`).
2. Provide the query string where each term is followed by a tilde (~) operator at the end of each whole term (`search=<string>~`). An expansion graph is created for every term in the query input.

Include an optional parameter, a number between 0 and 2 (default), if you want to specify the edit distance (~1). For example, "blue~" or "blue~1" would return "blue", "blues", and "glue".

Optionally, you can improve query performance by scoping the request to specific fields. Use the `searchFields` parameter to specify which fields to search. You can also use the `select` property to specify which fields are returned in the query response.

Testing fuzzy search

For simple testing, we recommend [Search explorer](#) or a [REST client](#) for iterating over a query expression. Both tools are interactive, which means you can quickly step through multiple variants of a term and evaluate the responses that come back.

When results are ambiguous, [hit highlighting](#) can help you identify the match in the response.

(!) Note

The use of hit highlighting to identify fuzzy matches has limitations and only works for basic fuzzy search. If your index has scoring profiles, or if you layer the query with more syntax, hit highlighting might fail to identify the match.

Example 1: fuzzy search with the exact term

Assume the following string exists in a "Description" field in a search document: "Test queries with special characters, plus strings for MSFT, SQL and Java."

Start with a fuzzy search on "special" and add hit highlighting to the Description field:

Console

```
search=special~&highlight=Description
```

In the response, because you added hit highlighting, formatting is applied to "special" as the matching term.

Output

```
"@search.highlights": {  
  "Description": [  
    "Test queries with <em>special</em> characters, plus strings for MSFT, SQL  
    and Java."  
  ]  
}
```

Try the request again, misspelling "special" by taking out several letters ("pe"):

Console

```
search=scial~&highlight=Description
```

So far, no change to the response. Given the default of 2 degrees distance, removing two characters "pe" from "special" still allows for a successful match on that term.

Output

```
"@search.highlights": {
    "Description": [
        "Test queries with <em>special</em> characters, plus strings for MSFT, SQL
        and Java."
    ]
}
```

Trying one more request, further modify the search term by taking out one last character for a total of three deletions (from "special" to "scal"):

Console

```
search=scal~&highlight=Description
```

Notice that the same response is returned, but now instead of matching on "special", the fuzzy match is on "SQL".

Output

```
"@search.score": 0.4232868,
"@search.highlights": {
    "Description": [
        "Mix of special characters, plus strings for MSFT, <em>SQL</em>, 2019,
        Linux, Java."
    ]
}
```

The point of this expanded example is to illustrate the clarity that hit highlighting can bring to ambiguous results. In all cases, the same document is returned. Had you relied on document IDs to verify a match, you might have missed the shift from "special" to "SQL".

See also

- [How full text search works in Azure AI Search \(query parsing architecture\)](#)
- [Search explorer](#)
- [How to query in .NET](#)
- [How to query in REST](#)

Create a hybrid query in Azure AI Search

08/28/2025

Hybrid search combines text (keyword) and vector queries in a single search request. Both queries execute in parallel. The results are merged and reordered by new search scores, using Reciprocal Rank Fusion (RRF) to return a unified result set. In many cases, [per benchmark tests](#), hybrid queries with semantic ranking return the most relevant results.

In this article, learn how to:

- Set up a basic hybrid request
- Add parameters and filters
- Improve relevance using semantic ranking or vector weights
- Optimize query behaviors by controlling inputs (`maxTextRecallSize`)

Prerequisites

- A search index containing `searchable` vector and nonvector fields. We recommend the [Import data \(new\)](#) wizard to create an index quickly. Otherwise, see [Create an index](#) and [Add vector fields to a search index](#).
- (Optional) If you want the [semantic ranker](#), your search service must be Basic tier or higher, with [semantic ranker enabled](#).
- (Optional) If you want built-in text-to-vector conversion of a query string, [create and assign a vectorizer](#) to vector fields in the search index.

Choose an API or tool

- Search Explorer in the Azure portal (supports both stable and preview API search syntax) has a JSON view that lets you paste in a hybrid request.
- Newer stable or preview packages of the Azure SDKs (see change logs for SDK feature support).
- [Stable REST APIs](#) or a recent preview API version if you're using preview features like `maxTextRecallSize` and `countAndFacetMode(preview)`.

For readability, we use REST examples to explain how the APIs work. You can use a REST client like Visual Studio Code with the REST extension to build hybrid queries. You can also use the Azure SDKs. For more information, see [Quickstart: Vector search](#).

Set up a hybrid query

This section explains the basic structure of a hybrid query and how to set one up in either Search Explorer or for execution in a REST client.

Results are returned in plain text, including vectors in fields marked as `retrievable`. Because numeric vectors aren't useful in search results, choose other fields in the index as a proxy for the vector match. For example, if an index has "descriptionVector" and "descriptionText" fields, the query can match on "descriptionVector" but the search result can show "descriptionText". Use the `select` parameter to specify only human-readable fields in the results.

Azure portal

1. Sign in to the [Azure portal](#) and find your search service.
2. Under **Search management > Indexes**, select an index that has vectors and non-vector content. [Search Explorer](#) is the first tab.
3. Under **View**, switch to **JSON view** so that you can paste in a vector query.
4. Replace the default query template with a hybrid query. A basic hybrid query has a text query specified in `search`, and a vector query specified under `vectorQueries.vector`. The text query and vector query can be equivalent or divergent, but it's common for them to share the same intent.

This example is from the [vector quickstart](#) that has vector and nonvector content, and several query examples. For brevity, the vector is truncated in this article.

JSON

```
{  
  "search": "historic hotel walk to restaurants and shopping",  
  "vectorQueries": [  
    {  
      "vector": [0.01944167, 0.0040178085, -0.007816401 ...  
      <remaining values omitted> ],  
      "k": 7,  
      "fields": "DescriptionVector",  
      "kind": "vector",  
      "exhaustive": true  
    }  
  ]  
}
```

5. Select **Search**.

💡 Tip

Search results are easier to read if you hide the vectors. In **Query Options**, turn on **Hide vector values in search results**.

6. Here's another version of the query. This one adds a `count` for the number of matches found, a `select` parameter for choosing specific fields, and a `top` parameter to return the top seven results.

JSON

```
{  
    "count": true,  
    "search": "historic hotel walk to restaurants and shopping",  
    "select": "HotelId, HotelName, Category, Tags, Description",  
    "top": 7,  
    "vectorQueries": [  
        {  
            "vector": [0.01944167, 0.0040178085, -0.007816401 ...  
<remaining values omitted> ],  
            "k": 7,  
            "fields": "DescriptionVector",  
            "kind": "vector",  
            "exhaustive": true  
        }  
    ]  
}
```

Set `maxTextRecallSize` and `countAndFacetMode`

⚠ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

A hybrid query can be tuned to control how much of each subquery contributes to the combined results. Setting `maxTextRecallSize` specifies how many BM25-ranked results are passed to the hybrid ranking model.

If you use `maxTextRecallSize`, you might also want to set `CountAndFacetMode`. This parameter determines whether the `count` and `facets` should include all documents that matched the search query, or only those documents retrieved within the `maxTextRecallSize` window. The default value is "countAllResults".

We recommend the [latest preview REST API](#) for setting these options.

💡 Tip

Another approach for hybrid query tuning is [vector weighting](#), used to increase the importance of vector queries in the request.

1. Use [Search - POST \(preview\)](#) or [Search - GET \(preview\)](#) to specify preview parameters.
2. Add a `hybridSearch` query parameter object to set the maximum number of documents recalled through the BM25-ranked results of a hybrid query. It has two properties:
 - `maxTextRecallSize` specifies the number of BM25-ranked results to provide to the Reciprocal Rank Fusion (RRF) ranker used in hybrid queries. The default is 1,000. The maximum is 10,000.
 - `countAndFacetMode` reports the counts for the BM25-ranked results (and for facets if you're using them). The default is all documents that match the query. Optionally, you can scope "count" to the `maxTextRecallSize`.
3. Set `maxTextRecallSize`:
 - Decrease `maxTextRecallSize` if vector similarity search is generally outperforming the text-side of the hybrid query.
 - Increase `maxTextRecallSize` if you have a large index, and the default isn't capturing a sufficient number of results. With a larger BM25-ranked result set, you can also set `top`, `skip`, and `next` to retrieve portions of those results.

The following REST examples show two use-cases for setting `maxTextRecallSize`.

The first example reduces `maxTextRecallSize` to 100, limiting the text side of the hybrid query to just 100 document. It also sets `countAndFacetMode` to include only those results from `maxTextRecallSize`.

HTTP

```
POST https://[service-name].search.windows.net/indexes/[index-name]/docs/search?  
api-version=2024-05-01-Preview
```

```
{  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [1.0, 2.0, 3.0],  
            "fields": "my_vector_field",  
            "k": 10  
        }  
    ],  
    "search": "hello world",  
    "hybridSearch": {  
        "maxTextRecallSize": 100,  
        "countAndFacetMode": "countRetrievableResults"  
    }  
}
```

The second example raises `maxTextRecallSize` to 5,000. It also uses top, skip, and next to pull results from large result sets. In this case, the request pulls in BM25-ranked results starting at position 1,500 through 2,000 as the text query contribution to the RRF composite result set.

HTTP

```
POST https://[service-name].search.windows.net/indexes/[index-name]/docs/search?  
api-version=2024-05-01-Preview
```

```
{  
    "vectorQueries": [  
        {  
            "kind": "vector",  
            "vector": [1.0, 2.0, 3.0],  
            "fields": "my_vector_field",  
            "k": 10  
        }  
    ],  
    "search": "hello world",  
    "top": 500,  
    "skip": 1500,  
    "next": 500,  
    "hybridSearch": {  
        "maxTextRecallSize": 5000,  
        "countAndFacetMode": "countRetrievableResults"  
    }  
}
```

Examples of hybrid queries

This section has multiple query examples that illustrate hybrid query patterns.

Example: Hybrid search with filter

This example adds a filter, which is applied to the `filterable` nonvector fields of the search index.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
Content-Type: application/json
api-key: {{admin-api-key}}
{
    "vectorQueries": [
        {
            "vector": [
                -0.009154141,
                0.018708462,
                ...
                -0.02178128,
                -0.00086512347
            ],
            "fields": "DescriptionVector",
            "kind": "vector",
            "k": 10
        }
    ],
    "search": "historic hotel walk to restaurants and shopping",
    "vectorFilterMode": "preFilter",
    "filter": "ParkingIncluded",
    "top": "10"
}
```

Key points:

- Filters are applied to the content of filterable fields. In this example, the `ParkingIncluded` field is a boolean and it's marked as `filterable` in the index schema.
- In hybrid queries, filters can be applied before query execution to reduce the query surface or after query execution to trim results. `"preFilter"` is the default. To use `postFilter` or `strictPostFilter` (preview), set the `filter processing mode` as shown in this example.
- When you postfilter query results, the number of results might be less than top-n.

Example: Hybrid search with filters targeting vector subqueries (preview)

Using the [latest preview REST API](#), you can override a global filter on the search request by applying a secondary filter that targets just the vector subqueries in a hybrid request.

This feature provides fine-grained control by ensuring that filters only influence the vector search results, leaving keyword-based search results unaffected.

The targeted filter fully overrides the global filter, including any filters used for [security trimming](#) or geospatial search. In cases where global filters are required, such as security trimming, you must explicitly include these filters in both the top-level filter and in each vector-level filter to ensure security and other constraints are consistently enforced.

To apply targeted vector filters:

- Use the [latest preview Search Documents REST API](#) or an Azure SDK beta package that provides the feature.
- Modify a query request, adding a new `vectorQueries.filterOverride` parameter set to an [OData filter expression](#).

Here's an example of hybrid query that adds a filter override. The global filter "Rating gt 3" is replaced at run time by the `filterOverride`.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-08-01-preview
```

```
{
  "vectorQueries": [
    {
      "vector": [
        -0.009154141,
        0.018708462,
        ...
        -0.02178128,
        -0.00086512347
      ],
      "fields": "DescriptionVector",
      "kind": "vector",
      "exhaustive": true,
      "filterOverride": "Address/City eq 'Seattle'",
      "k": 10
    }
  ],
  "search": "historic hotel walk to restaurants and shopping",
  "select": "HotelName, Description, Address/City, Rating",
  "filter": "Rating gt 3"
  "debug": "vector",
```

```
    "top": 10
}
```

Example: Semantic hybrid search

Assuming that you [have semantic ranker](#) and your index definition includes a [semantic configuration](#), you can formulate a query that includes vector search and keyword search, with semantic ranking over the merged result set. Optionally, you can add captions and answers.

Whenever you use semantic ranking with vectors, make sure `k` is set to 50. Semantic ranker uses up to 50 matches as input. Specifying less than 50 deprives the semantic ranking models of necessary inputs.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
Content-Type: application/json
api-key: {{admin-api-key}}
{
  "vectorQueries": [
    {
      "vector": [
        -0.009154141,
        0.018708462,
        ...
        -0.02178128,
        -0.00086512347
      ],
      "fields": "DescriptionVector",
      "kind": "vector",
      "k": 50
    }
  ],
  "search": "historic hotel walk to restaurants and shopping",
  "select": "HotelName, Description, Tags",
  "queryType": "semantic",
  "semanticConfiguration": "my-semantic-config",
  "captions": "extractive",
  "answers": "extractive",
  "top": "50"
}
```

Key points:

- Semantic ranker accepts up to 50 results from the merged response.
- "queryType" and "semanticConfiguration" are required.

- "captions" and "answers" are optional. Values are extracted from verbatim text in the results. An answer is only returned if the results include content having the characteristics of an answer to the query.

Example: Semantic hybrid search with filter

Here's the last query in the collection. It's the same semantic hybrid query as the previous example, but with a filter.

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
Content-Type: application/json
api-key: {{admin-api-key}}
{
  "vectorQueries": [
    {
      "vector": [
        -0.009154141,
        0.018708462,
        . . .
        -0.02178128,
        -0.00086512347
      ],
      "fields": "DescriptionVector",
      "kind": "vector",
      "k": 50
    }
  ],
  "search": "historic hotel walk to restaurants and shopping",
  "select": "HotelName, Description, Tags",
  "queryType": "semantic",
  "semanticConfiguration": "my-semantic-config",
  "captions": "extractive",
  "answers": "extractive",
  "filter": "ParkingIncluded",
  "vectorFilterMode": "preFilter",
  "top": "50"
}
```

Key points:

- The filter mode can affect the number of results available to the semantic reranker. As a best practice, it's smart to give the semantic ranker the maximum number of documents (50). If prefilters or postfilters are too selective, you might be underserving the semantic ranker by giving it fewer than 50 documents to work with.

- `preFilter` is applied before query execution. If prefilter reduces the search area to 100 documents, the vector query executes over the `DescriptionVector` field for those 100 documents, returning the k=50 best matches. Those 50 matching documents then pass to RRF for merged results, and then to semantic ranker.
- `postFilter` is applied after query execution. If k=50 returns 50 matches on the vector query side, followed by a post-filter applied to the 50 matches, your results are reduced by the number of documents that meet filter criteria. This leaves you with fewer than 50 documents to pass to semantic ranker. Keep this in mind if you're using semantic ranking. The semantic ranker works best if it has 50 documents as input.
- `strictPostFilter` (preview) is applied on the unfiltered top-`k` results after query execution. It always returns less than or equal to `k` documents. If the unfiltered k=50 returns 50 unfiltered results, and the filter matches 30 documents, only 30 documents are returned in the result set, even if the index has more than 30 documents that match the filter. Since this mode has the greatest reduction in recall, we don't recommend that you use it with semantic ranker.

Configure a query response

When you're setting up the hybrid query, think about the response structure. The search engine ranks the matching documents and returns the most relevant results. The response is a flattened rowset. Parameters on the query determine which fields are in each row and how many rows are in the response.

Fields in a response

Search results are composed of `retrievable` fields from your search index. A result is either:

- All `retrievable` fields (a REST API default).
- Fields explicitly listed in a `select` parameter on the query.

The examples in this article used a `select` statement to specify text (nonvector) fields in the response.

Note

Vectors aren't reverse engineered into human readable text, so avoid returning them in the response. Instead, choose nonvector fields that are representative of the search

document. For example, if the query targets a "DescriptionVector" field, return an equivalent text field if you have one ("Description") in the response.

Number of results

A query might match to any number of documents, as many as all of them if the search criteria are weak (for example "search=*" for a null query). Because it's seldom practical to return unbounded results, you should specify a maximum for the *overall response*:

- `"top": n` results for keyword-only queries (no vector)
- `"k": n` results for vector-only queries
- `"top": n` results for hybrid queries (with or without semantic) that include a "search" parameter

Both `k` and `top` are optional. Unspecified, the default number of results in a response is 50.

You can set `top` and `skip` to [page through more results](#) or change the default.

(!) Note

If you're using hybrid search in 2024-05-01-preview API, you can control the number of results from the keyword query using [`maxTextRecallSize`](#). Combine this with a setting for `k` to control the representation from each search subsystem (keyword and vector).

Semantic ranker results

(!) Note

The semantic ranker can take up to 50 results.

If you're using semantic ranker in 2024-05-01-preview or later, it's a best practice to set `k` and `maxTextRecallSize` to sum to at least 50 total. You can then restrict the results returned to the user with the `top` parameter.

If you're using semantic ranker in previous APIs do the following:

- For keyword-only search (no vectors) set `top` to 50
- For hybrid search set `k` to 50, to ensure that the semantic ranker gets at least 50 results.

Ranking

Multiple sets are created for hybrid queries, with or without the optional [semantic reranking](#). Ranking of results is computed by Reciprocal Rank Fusion (RRF).

In this section, compare the responses between single vector search and simple hybrid search for the top result. The different ranking algorithms, HNSW's similarity metric and RRF is this case, produce scores that have different magnitudes. This behavior is by design. RRF scores can appear quite low, even with a high similarity match. Lower scores are a characteristic of the RRF algorithm. In a hybrid query with RRF, more of the reciprocal of the ranked documents are included in the results, given the relatively smaller score of the RRF ranked documents, as opposed to pure vector search.

Single Vector Search: @search.score for results ordered by cosine similarity (default vector similarity distance function).

JSON

```
{  
  "@search.score": 0.8399121,  
  "HotelId": "49",  
  "HotelName": "Swirling Currents Hotel",  
  "Description": "Spacious rooms, glamorous suites and residences, rooftop pool,  
 walking access to shopping, dining, entertainment and the city center.",  
  "Category": "Luxury",  
  "Address": {  
    "City": "Arlington"  
  }  
}
```

Hybrid Search: @search.score for hybrid results ranked using Reciprocal Rank Fusion.

JSON

```
{  
  "@search.score": 0.032786883413791656,  
  "HotelId": "49",  
  "HotelName": "Swirling Currents Hotel",  
  "Description": "Spacious rooms, glamorous suites and residences, rooftop pool,  
 walking access to shopping, dining, entertainment and the city center.",  
  "Category": "Luxury",  
  "Address": {  
    "City": "Arlington"  
  }  
}
```

Next step

We recommend reviewing vector demo code for [Python](#), [C#](#) or [JavaScript](#).

Relevance in keyword search (BM25 scoring)

08/27/2025

This article explains the BM25 relevance scoring algorithm used to compute search scores for [full text search](#). BM25 relevance applies to full text search only. Filter queries, autocomplete and suggested queries, wildcard search, and fuzzy search queries aren't scored or ranked for relevance.

Scoring algorithms used in full text search

Azure AI Search provides the following scoring algorithms for full text search:

 [Expand table](#)

Algorithm	Usage	Range
BM25Similarity	Fixed algorithm on all search services created after July 2020. You can configure this algorithm, but you can't switch to an older one (classic).	Unbounded.
ClassicSimilarity	Default on older search services that predate July 2020. On older services, you can opt-in for BM25 and choose a the BM25 algorithm on a per-index basis.	0 < 1.00

Both BM25 and Classic are TF-IDF-like retrieval functions that use the term frequency (TF) and the inverse document frequency (IDF) as variables to calculate relevance scores for each document-query pair, which is then used for ranking results. While conceptually similar to classic, BM25 is rooted in probabilistic information retrieval that produces more intuitive matches, as measured by user research.

BM25 offers [advanced customization options](#), such as allowing the user to decide how the relevance score scales with the term frequency of matched terms.

How BM25 ranking works

Relevance scoring refers to the computation of a search score (`@search.score`) that serves as an indicator of an item's relevance in the context of the current query. The range is unbounded. However, the higher the score, the more relevant the item.

The search score is computed based on statistical properties of the string input and the query itself. Azure AI Search finds documents that match on search terms (some or all, depending on `searchMode`), favoring documents that contain many instances of the search term. The search score goes up even higher if the term is rare across the data index, but common within the document. The basis for this approach to computing relevance is known as *TF-IDF* or term frequency-inverse document frequency.

Search scores can be repeated throughout a result set. When multiple hits have the same search score, the ordering of the same scored items is undefined and not stable. Run the query again, and you might see items shift position, especially if you're using the free service or a billable service with multiple replicas. Given two items with an identical score, there's no guarantee that one appears first.

To break the tie among repeating scores, you can add an `$orderby clause` to first order by score, then order by another sortable field (for example, `$orderby=search.score() desc, Rating desc`).

Only fields marked as `searchable` in the index, or `searchFields` in the query, are used for scoring. Only fields marked as `retrievable`, or fields specified in `select` in the query, are returned in search results, along with their search score.

! Note

A `@search.score = 1` indicates an un-scored or un-ranked result set. The score is uniform across all results. Un-scored results occur when the query form is fuzzy search, wildcard or regex queries, or an empty search (`search=*`, sometimes paired with filters, where the filter is the primary means for returning a match).

The following video segment fast-forwards to an explanation of the generally available ranking algorithms used in Azure AI Search. You can watch the full video for more background.

https://www.youtube-nocookie.com/embed/Y_X6USgvB1g?

[version=3&start=322&end=643 ↗](#)

Scores in a text results

Whenever results are ranked, `@search.score` property contains the value used to order the results.

The following table identifies the scoring property, algorithm, and range.

Search method	Parameter	Scoring algorithm	Range
full text search	@search.score	BM25 algorithm, using the parameters specified in the index .	Unbounded.

Score variation

Search scores convey general sense of relevance, reflecting the strength of match relative to other documents in the same result set. But scores aren't always consistent from one query to the next, so as you work with queries, you might notice small discrepancies in how search documents are ordered. There are several explanations for why this might occur.

Cause	Description
Identical scores	If multiple documents have the same score, any one of them might appear first.
Data volatility	Index content varies as you add, modify, or delete documents. Term frequencies will change as index updates are processed over time, affecting the search scores of matching documents.
Multiple replicas	For services using multiple replicas, queries are issued against each replica in parallel. The index statistics used to calculate a search score are calculated on a per-replica basis, with results merged and ordered in the query response. Replicas are mostly mirrors of each other, but statistics can differ due to small differences in state. For example, one replica might have deleted documents contributing to their statistics, which were merged out of other replicas. Typically, differences in per-replica statistics are more noticeable in smaller indexes. The following section provides more information about this condition.

Sharding effects on query results

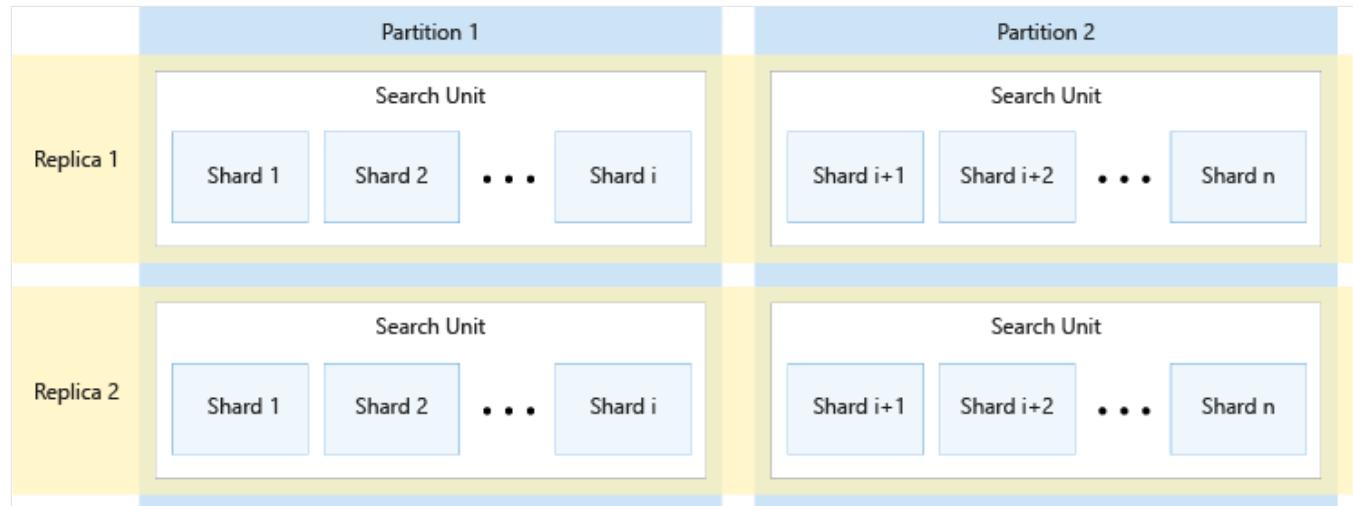
A *shard* is a chunk of an index. Azure AI Search subdivides an index into *shards* to make the process of adding partitions faster (by moving shards to new search units). On a search service, shard management is an implementation detail and nonconfigurable, but knowing that an index is sharded helps to understand the occasional anomalies in ranking and autocomplete behaviors:

- **Ranking anomalies:** Search scores are computed at the shard level first, and then aggregated up into a single result set. Depending on the characteristics of shard content,

matches from one shard might be ranked higher than matches in another one. If you notice counter intuitive rankings in search results, it's most likely due to the effects of sharding, especially if indexes are small. You can avoid these ranking anomalies by choosing to [compute scores globally across the entire index](#), but doing so will incur a performance penalty.

- Autocomplete anomalies: Autocomplete queries, where matches are made on the first several characters of a partially entered term, accept a fuzzy parameter that forgives small deviations in spelling. For autocomplete, fuzzy matching is constrained to terms within the current shard. For example, if a shard contains "Microsoft" and a partial term of "micro" is entered, the search engine will match on "Microsoft" in that shard, but not in other shards that hold the remaining parts of the index.

The following diagram shows the relationship between replicas, partitions, shards, and search units. It shows an example of how a single index is spanned across four search units in a service with two replicas and two partitions. Each of the four search units stores only half of the shards of the index. The search units in the left column store the first half of the shards, comprising the first partition, while those in the right column store the second half of the shards, comprising the second partition. Since there are two replicas, there are two copies of each index shard. The search units in the top row store one copy, comprising the first replica, while those in the bottom row store another copy, comprising the second replica.



The diagram above is only one example. Many combinations of partitions and replicas are possible, up to a maximum of 36 total search units.

! Note

The number of replicas and partitions divides evenly into 12 (specifically, 1, 2, 3, 4, 6, 12). Azure AI Search pre-divides each index into 12 shards so that it can be spread in equal portions across all partitions. For example, if your service has three partitions and you create an index, each partition will contain four shards of the index. How Azure AI Search

shards an index is an implementation detail, subject to change in future releases. Although the number is 12 today, you shouldn't expect that number to always be 12 in the future.

Scoring statistics and sticky sessions

For scalability, Azure AI Search distributes each index horizontally through a sharding process, which means that [portions of an index are physically separate](#).

By default, the score of a document is calculated based on statistical properties of the data *within a shard*. This approach is generally not a problem for a large corpus of data, and it provides better performance than having to calculate the score based on information across all shards. That said, using this performance optimization could cause two very similar documents (or even identical documents) to end up with different relevance scores if they end up in different shards.

If you prefer to compute the score based on the statistical properties across all shards, you can do so by adding `scoringStatistics=global` as a [query parameter](#) (or add `"scoringStatistics": "global"` as a body parameter of the [query request](#)).

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?api-version=2025-09-01
{
    "search": "<query string>",
    "scoringStatistics": "global"
}
```

Using `scoringStatistics` will ensure that all shards in the same replica provide the same results. That said, different replicas can be slightly different from one another as they're always getting updated with the latest changes to your index. In some scenarios, you might want your users to get more consistent results during a "query session". In such scenarios, you can provide a `sessionId` as part of your queries. The `sessionId` is a unique string that you create to refer to a unique user session.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?api-version=2025-09-01
{
    "search": "<query string>",
    "sessionId": "<string>"
}
```

As long as the same `sessionId` is used, a best-effort attempt is made to target the same replica, increasing the consistency of results your users will see.

 Note

Reusing the same `sessionId` values repeatedly can interfere with the load balancing of the requests across replicas and adversely affect the performance of the search service. The value used as sessionId can't start with a '_' character.

Relevance tuning

In Azure AI Search, for keyword search and the text portion of a hybrid query, you can configure BM25 algorithm parameters plus tune search relevance and boost search scores through the following mechanisms.

 Expand table

Approach	Implementation	Description
BM25 algorithm configuration	Search index	Configure how document length and term frequency affect the relevance score.
Scoring profiles	Search index	Provide criteria for boosting the search score of a match based on content characteristics. For example, you can boost matches based on their revenue potential, promote newer items, or perhaps boost items that have been in inventory too long. A scoring profile is part of the index definition, composed of weighted fields, functions, and parameters. You can update an existing index with scoring profile changes, without incurring an index rebuild.
Semantic ranking	Query request	Applies machine reading comprehension to search results, promoting more semantically relevant results to the top.
featuresMode parameter	Query request	This parameter is mostly used for unpacking a BM25-ranked score, but it can be used for in code that provides a custom scoring solution .

featuresMode parameter (preview)

 Note

The `featuresMode` parameter isn't documented in the REST APIs, but you can use it on a preview REST API call to Search Documents for text (Keyword) search that's BM25-ranked.

Search Documents (preview) requests support a `featuresMode` parameter that provides more detail about a BM25 relevance score at the field level. Whereas the `@searchScore` is calculated for the document all-up (how relevant is this document in the context of this query), `featuresMode` reveals information about individual fields, as expressed in a `@search.features` structure. The structure contains all fields used in the query (either specific fields through `searchFields` in a query, or all fields attributed as `searchable` in an index).

Valid values for `featuresMode`:

- "none" (default). No feature-level scoring details are returned.
- "enabled". Returns detailed scoring breakdowns per field

For each field, `@search.features` give you the following values:

- Number of unique tokens found in the field
- Similarity score, or a measure of how similar the content of the field is, relative to the query term
- Term frequency, or the number of times the query term was found in the field

This parameter is especially useful when you're trying to understand why certain documents rank higher or lower in search results. It helps explain how different fields contribute to the overall score.

For a query that targets a "description" field, a request might look like this:

HTTP

```
POST {{baseUrl}}/indexes/hotels-sample-index/docs/search?api-version=2025-08-01-  
preview HTTP/1.1  
Content-Type: application/json  
Authorization: Bearer {{accessToken}}  
  
{  
    "search": "lake view",  
    "select": "HotelId, HotelName, Tags, Description",  
    "featuresMode": "enabled",  
    "searchFields": "Description, Tags",  
    "count": true  
}
```

A response that includes `@search.features` might look like the following example.

JSON

```
"value": [
  {
    "@search.score": 3.0860271,
    "@search.features": {
      "Description": {
        "uniqueTokenMatches": 2.0,
        "similarityScore": 3.0860272,
        "termFrequency": 2.0
      }
    },
    "HotelName": "Downtown Mix Hotel",
    "Description": "Mix and mingle in the heart of the city. Shop and dine, mix and mingle in the heart of downtown, where fab lake views unite with a cheeky design.",
    "Tags": [
      "air conditioning",
      "laundry service",
      "free wifi"
    ]
  },
  {
    "@search.score": 2.7294855,
    "@search.features": {
      "Description": {
        "uniqueTokenMatches": 1.0,
        "similarityScore": 1.6023184,
        "termFrequency": 1.0
      },
      "Tags": {
        "uniqueTokenMatches": 1.0,
        "similarityScore": 1.1271671,
        "termFrequency": 1.0
      }
    },
    "HotelName": "Ocean Water Resort & Spa",
    "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near the pier, rooftop pool, waterfront dining & more.",
    "Tags": [
      "view",
      "pool",
      "restaurant"
    ]
  }
]
```

You can consume these data points in [custom scoring solutions](#) or use the information to debug search relevance problems.

Number of ranked results in a full text query response

By default, if you aren't using pagination, the search engine returns the top 50 highest ranking matches for full text search. You can use the `top` parameter to return a smaller or larger number of items (up to 1,000 in a single response). You can use `skip` and `next` to page results. Paging determines the number of results on each logical page and supports content navigation. For more information, see [Shape search results](#).

If your full text query is part of a [hybrid query](#), you can [set maxTextRecallSize](#) to increase or decrease the number of results from the text side of the query.

Full text search is subject to a maximum limit of 1,000 matches (see [API response limits](#)). Once 1,000 matches are found, the search engine no longer looks for more.

See also

- [Scoring Profiles](#)
- [REST API Reference](#)
- [Search Documents API](#)
- [Azure AI Search .NET SDK](#)

Configure BM25 relevance scoring

Article • 02/24/2025

In this article, learn how to configure the [BM25 relevance scoring algorithm](#) used by Azure AI Search for full text search queries. It also explains how to enable BM25 on older search services.

BM25 applies to:

- Queries that use the `search` parameter for full text search, on text fields having a `searchable` attribution.
- Scoring is scoped to `searchFields`, or to all `searchable` fields if `searchFields` is null.

The search engine uses BM25 to calculate a `@searchScore` for each match in a given query. Matching documents are ranked by their search score, with the top results returned in the query response. It's possible to get some [score variation](#) in results, even from the same query executing over the same search index, but usually these variations are small and don't change the overall ranking of results.

BM25 has defaults for weighting term frequency and document length. You can customize these properties if the defaults aren't suited to your content. Configuration changes are scoped to individual indexes, which means you can adjust relevance scoring based on the characteristics of each index.

Default scoring algorithm

Depending on the age of your search service, Azure AI Search supports two [scoring algorithms](#) for a full text search query:

- Okapi BM25 algorithm (after July 15, 2020)
- Classic similarity algorithm (before July 15, 2020)

BM25 ranking is the default because it tends to produce search rankings that align better with user expectations. It includes [parameters](#) for tuning results based on factors such as document size. For search services created after July 2020, BM25 is the only scoring algorithm. If you try to set "similarity" to `ClassicSimilarity` on a new service, an HTTP 400 error is returned because that algorithm isn't supported by the service.

For older services, classic similarity remains the default algorithm. Older services can [upgrade to BM25](#) on a per-index basis. When switching from classic to BM25, you can expect to see some differences how search results are ordered.

Set BM25 parameters

BM25 ranking provides two parameters for tuning the relevance score calculation.

1. Use a [Create or Update Index](#) request to set BM25 parameters:

HTTP

```
PUT [service-name].search.windows.net/indexes/[index-name]?api-version=2024-07-01&allowIndexDowntime=true
{
    "similarity": {
        "@odata.type": "#Microsoft.Azure.Search.BM25Similarity",
        "b" : 0.75,
        "k1" : 1.2
    }
}
```

2. If the index is live, append the `allowIndexDowntime=true` URI parameter on the request, shown on the previous example.

Because Azure AI Search doesn't allow updates to a live index, you need to take the index offline so that the parameters can be added. Indexing and query requests fail while the index is offline. The duration of the outage is the amount of time it takes to update the index, usually no more than several seconds. When the update is complete, the index comes back automatically.

3. Set `"b"` and `"k1"` to custom values, and then send the request.

[+] [Expand table](#)

Property	Type	Description
k1	number	<p>Controls the scaling function between the term frequency of each matching terms to the final relevance score of a document-query pair. Values are usually 0.0 to 3.0, with 1.2 as the default.</p> <p>A value of 0.0 represents a "binary model", where the contribution of a single matching term is the same for all matching documents, regardless of how many times that term appears in the text. Larger k1 values allow the score to continue to increase as more instances of the same term is found in the document.</p> <p>Using a larger k1 value is important in cases where multiple terms are included in a search query. In those cases, you might want to favor documents matching more query terms, over documents that only match a single term, multiple times. For example, when querying for</p>

Property	Type	Description
		<p>the terms "Apollo Spaceflight", you might want to lower the score of an article about Greek Mythology that contains the term "Apollo" a few dozen times, without mentions of "Spaceflight", relative to another article that explicitly mentions both "Apollo" and "Spaceflight" a handful of times only.</p>
b	number	<p>Controls how the length of a document affects the relevance score. Values are between 0 and 1, with 0.75 as the default.</p> <p>A value of 0.0 means the length of the document doesn't influence the score. A value of 1.0 means the effect of term frequency on relevance score is normalized by the document's length.</p> <p>Normalizing the term frequency by the document's length is useful in cases where you want to penalize longer documents. In some cases, longer documents (such as a complete novel), are more likely to contain many irrelevant terms, compared to shorter documents.</p>

Enable BM25 scoring on older services

If you're running a search service that was created from March 2014 through July 15, 2020, you can enable BM25 by setting a "similarity" property on new indexes. The property is only exposed on new indexes, so if you want BM25 on an existing index, you must drop and [rebuild the index](#) with a "similarity" property set to

`Microsoft.Azure.Search.BM25Similarity`.

Once an index exists with a "similarity" property, you can switch between `BM25Similarity` or `ClassicSimilarity`.

The following links describe the Similarity property in the Azure SDKs.

[\[+\] Expand table](#)

Client library	Similarity property
.NET	SearchIndex.Similarity
Java	SearchIndex.setSimilarity
JavaScript	SearchIndex.Similarity
Python	similarity property on SearchIndex

REST example

You can also use the [REST API](#). The following example creates a new index with the "similarity" property set to BM25:

```
HTTP  
  
PUT [service-name].search.windows.net/indexes/[index name]?api-version=2024-07-01  
{  
    "name": "indexName",  
    "fields": [  
        {  
            "name": "id",  
            "type": "Edm.String",  
            "key": true  
        },  
        {  
            "name": "name",  
            "type": "Edm.String",  
            "searchable": true,  
            "analyzer": "en.lucene"  
        },  
        ...  
    ],  
    "similarity": {  
        "@odata.type": "#Microsoft.Azure.Search.BM25Similarity"  
    }  
}
```

See also

- [Relevance and scoring in Azure AI Search](#)
- [REST API Reference](#)
- [Add scoring profiles to your index](#)
- [Create Index API](#)
- [Azure AI Search .NET SDK](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Relevance in vector search

07/09/2025

During vector query execution, the search engine looks for similar vectors to find the best candidates to return in search results. Depending on how you indexed the vector content, the search for relevant matches is either exhaustive, or constrained to nearest neighbors for faster processing. Once candidates are found, similarity metrics are used to score each result based on the strength of the match.

This article explains the algorithms used to find relevant matches and the similarity metrics used for scoring. It also offers tips for improving relevance if search results don't meet expectations.

Algorithms used in vector search

Vector search algorithms include:

- [Exhaustive K-Nearest Neighbors \(KNN\)](#), which performs a brute-force scan of the entire vector space.
- [Hierarchical Navigable Small World \(HNSW\)](#), which performs an [Approximate Nearest Neighbor \(ANN\)](#) search.

Only vector fields marked as `searchable` in the index or `searchFields` in the query are used for searching and scoring.

About exhaustive KNN

Exhaustive KNN calculates the distances between all pairs of data points and finds the exact `k` nearest neighbors for a query point. Because the algorithm doesn't require fast random access of data points, KNN doesn't consume [vector index size](#) quota. However, it provides the global set of nearest neighbors.

Exhaustive KNN is computationally intensive, so use it for small to medium datasets or when the need for precision outweighs the need for query performance. Another use case is building a dataset to evaluate the recall of an [ANN algorithm](#), as exhaustive KNN can be used to build the ground truth set of nearest neighbors.

About HNSW

HNSW is an ANN algorithm optimized for high-recall, low-latency applications with unknown or volatile data distribution. During indexing, HNSW creates extra data structures that organize data points into a hierarchical graph. During query execution, HNSW navigates through this graph to find the most relevant matches, enabling efficient nearest neighbor searches.

HNSW requires all data points to reside in memory for fast random access, which consumes [vector index size](#) quota. This design balances search accuracy with computational efficiency and makes HNSW suitable for most scenarios, especially when searching over larger datasets.

HNSW offers several tunable configuration parameters to optimize throughput, latency, and recall for your search application. For example, fields that specify HNSW also support exhaustive KNN using the [query request](#) parameter `"exhaustive": true`. However, fields indexed for `exhaustiveKnn` don't support HNSW queries because the extra data structures that enable efficient search don't exist.

About ANN

ANN is a class of algorithms for finding matches in vector space. This class of algorithms uses different data structures or data partitioning methods to significantly reduce the search space and accelerate query processing.

ANN algorithms sacrifice some accuracy but offer scalable and faster retrieval of approximate nearest neighbors, which makes them ideal for balancing accuracy and efficiency in modern information retrieval applications. You can adjust the parameters of your algorithm to fine-tune the recall, latency, memory, and disk footprint requirements of your search application.

Azure AI Search uses HNSW for its ANN algorithm.

How nearest neighbor search works

Vector queries execute against an embedding space consisting of vectors generated from the same embedding model. Generally, the input value within a query request is fed into the same machine learning model that generated embeddings in the vector index. The output is a vector in the same embedding space. Since similar vectors are clustered close together, finding matches is equivalent to finding the vectors that are closest to the query vector, and returning the associated documents as the search result.

For example, if a query request is about hotels, the model maps the query into a vector that exists somewhere in the cluster of vectors representing documents about hotels. Identifying which vectors are the most similar to the query, based on a similarity metric, determines which documents are the most relevant.

When vector fields are indexed for exhaustive KNN, the query executes against "all neighbors". For fields indexed for HNSW, the search engine uses an HNSW graph to search over a subset of nodes within the vector index.

Creating the HNSW graph

During indexing, the search service constructs the HNSW graph. The goal of indexing a new vector into an HNSW graph is to add it to the graph structure in a manner that allows for efficient nearest neighbor search. The following steps summarize the process:

1. Initialization: Start with an empty HNSW graph, or the existing HNSW graph if it's not a new index.
2. Entry point: This is the top-level of the hierarchical graph and serves as the starting point for indexing.
3. Adding to the graph: Different hierarchical levels represent different granularities of the graph, with higher levels being more global, and lower levels being more granular. Each node in the graph represents a vector point.
 - Each node is connected to up to `m` neighbors that are nearby. This is the `m` parameter.
 - The number of data points considered as candidate connections is governed by the `efConstruction` parameter. This dynamic list forms the set of closest points in the existing graph for the algorithm to consider. Higher `efConstruction` values result in more nodes being considered, which often leads to denser local neighborhoods for each vector.
 - These connections use the configured similarity `metric` to determine distance. Some connections are "long-distance" connections that connect across different hierarchical levels, creating shortcuts in the graph that enhance search efficiency.
4. Graph pruning and optimization: This can happen after indexing all vectors, and it improves navigability and efficiency of the HNSW graph.

Navigating the HNSW graph at query time

A vector query navigates the hierarchical graph structure to scan for matches. The following steps summarize the steps in the process:

1. Initialization: The algorithm initiates the search at the top-level of the hierarchical graph. This entry point contains the set of vectors that serve as starting points for search.

2. Traversal: Next, it traverses the graph level by level, navigating from the top-level to lower levels, selecting candidate nodes that are closer to the query vector based on the configured distance metric, such as cosine similarity.
3. Pruning: To improve efficiency, the algorithm prunes the search space by only considering nodes that are likely to contain nearest neighbors. This is achieved by maintaining a priority queue of potential candidates and updating it as the search progresses. The length of this queue is configured by the parameter `efSearch`.
4. Refinement: As the algorithm moves to lower, more granular levels, HNSW considers more neighbors near the query, which allows the candidate set of vectors to be refined, improving accuracy.
5. Completion: The search completes when the desired number of nearest neighbors have been identified, or when other stopping criteria are met. This desired number of nearest neighbors is governed by the query-time parameter `k`.

Similarity metrics used to measure nearness

The algorithm finds candidate vectors to evaluate similarity. To perform this task, a similarity metric calculation compares the candidate vector to the query vector and measures the similarity. The algorithm keeps track of the ordered set of most similar vectors that its found, which forms the ranked result set when the algorithm has reached completion.

[] Expand table

Metric	Description
<code>cosine</code>	This metric measures the angle between two vectors, and isn't affected by differing vector lengths. Mathematically, it calculates the angle between two vectors. Cosine is the similarity metric used by Azure OpenAI embedding models , so if you're using Azure OpenAI, specify <code>cosine</code> in the vector configuration.
<code>dotProduct</code>	This metric measures both the length of each pair of two vectors, and the angle between them. Mathematically, it calculates the products of vectors' magnitudes and the angle between them. For normalized vectors, this is identical to <code>cosine</code> similarity, but slightly more performant.
<code>euclidean</code>	(also known as <code>l2 norm</code>) This metric measures the length of the vector difference between two vectors. Mathematically, it calculates the Euclidean distance between two vectors, which is the l2-norm of the difference of the two vectors.

! Note

If you run two or more vector queries in parallel, or if you do a hybrid search that combines vector and text queries in the same request, [Reciprocal Rank Fusion \(RRF\)](#) is used for scoring the final search results.

Scores in a vector search results

Scores are calculated and assigned to each match, with the highest matches returned as `k` results. The `@search.score` property contains the score. The following table shows the range within which a score will fall.

 Expand table

Search method	Parameter	Scoring metric	Range
vector search	<code>@search.score</code>	Cosine	0.333 - 1.00

For `cosine` metric, it's important to note that the calculated `@search.score` isn't the cosine value between the query vector and the document vectors. Instead, Azure AI Search applies transformations such that the score function is monotonically decreasing, meaning score values will always decrease in value as the similarity becomes worse. This transformation ensures that search scores are usable for ranking purposes.

There are some nuances with similarity scores:

- Cosine similarity is defined as the cosine of the angle between two vectors.
- Cosine distance is defined as `1 - cosine_similarity`.

To create a monotonically decreasing function, the `@search.score` is defined as `1 / (1 + cosine_distance)`.

Developers who need a cosine value instead of the synthetic value can use a formula to convert the search score back to cosine distance:

C#

```
double ScoreToSimilarity(double score)
{
    double cosineDistance = (1 - score) / score;
    return -cosineDistance + 1;
}
```

Having the original cosine value can be useful in custom solutions that set up thresholds to trim results of low quality results.

Tips for relevance tuning

If you aren't getting relevant results, experiment with changes to [query configuration](#). There are no specific tuning features, such as a scoring profile or field or term boosting, for vector queries:

- Experiment with [chunk size and overlap](#). Try increasing the chunk size and ensuring there's sufficient overlap to preserve context or continuity between chunks.
- For HNSW, try different levels of `efConstruction` to change the internal composition of the proximity graph. The default is 400. The range is 100 to 1,000.
- Increase `k` results to feed more search results into a chat model, if you're using one.
- Try [hybrid queries](#) with semantic ranking. In benchmark testing, this combination consistently produced the most relevant results.

Next steps

- [Try the quickstart](#)
- [Create and configure a vector index](#)
- [Learn more about embeddings](#)
- [Learn more about data chunking](#)

Relevance scoring in hybrid search using Reciprocal Rank Fusion (RRF)

09/28/2025

Reciprocal Rank Fusion (RRF) is an algorithm that evaluates the search scores from multiple, previously ranked results to produce a unified result set. In Azure AI Search, RRF is used when two or more queries execute in parallel. Namely, for [hybrid queries](#) and for [multiple vector queries](#). Each individual query produces a ranked result set, and RRF merges and homogenizes the rankings into a single result set for the query response.

RRF is based on the concept of *reciprocal rank*, which is the inverse of the rank of the first relevant document in a list of search results. The goal of the technique is to take into account the position of the items in the original rankings, and give higher importance to items that are ranked higher in multiple lists. This can help improve the overall quality and reliability of the final ranking, making it more useful for the task of fusing multiple ordered search results.

How RRF ranking works

RRF works by taking the search results from multiple methods, assigning a reciprocal rank score to each document in the results, and then combining the scores to create a new ranking. The concept is that documents appearing in the top positions across multiple search methods are likely to be more relevant and should be ranked higher in the combined result.

Here's a simple explanation of the RRF process:

1. Obtain ranked search results from multiple queries executing in parallel.
2. Assign reciprocal rank scores for result in each of the ranked lists. RRF generates a new `@search.score` for each match in each result set. For each document in the search results, the engine assigns a reciprocal rank score based on its position in the list. The score is calculated as $1/(rank + k)$, where `rank` is the position of the document in the list, and `k` is a constant, which was experimentally observed to perform best if it's set to a small value like 60. **Note that this `k` value is a constant in the RRF algorithm and entirely separate from the `k` that controls the number of nearest neighbors.**
3. Combine scores. For each document, the engine sums the reciprocal rank scores obtained from each search system, producing a combined score for each document.
4. The engine ranks documents based on combined scores and sorts them. The resulting list is the fused ranking.

Only fields marked as `searchable` in the index, or `searchFields` in the query, are used for scoring. Only fields marked as `retrievable`, or fields specified in `select` in the query, are returned in search results, along with their search score.

Parallel query execution

RRF is used anytime there's more than one query execution. The following examples illustrate query patterns where parallel query execution occurs:

- A full text query, plus one vector query (simple hybrid scenario), equals two query executions.
- A full text query, plus one vector query targeting two vector fields, equals three query executions.
- A full text query, plus two vector queries targeting five vector fields, equals 11 query executions

Scores in a hybrid search results

Whenever results are ranked, `@search.score` property contains the value used to order the results. Scores are generated by ranking algorithms that vary for each method. Each algorithm has its own range and magnitude.

The following chart identifies the scoring property returned on each match, algorithm, and range of scores for each relevance ranking algorithm. For more information and a diagram of the scoring workflow, see [Relevance in Azure AI Search](#).

 Expand table

Search method	Parameter	Scoring algorithm	Range
full-text search	<code>@search.score</code>	BM25 algorithm	No upper limit.
vector search	<code>@search.score</code>	HNSW algorithm, using the similarity metric specified in the HNSW configuration.	0.333 - 1.00 (Cosine), 0 to 1 for Euclidean and DotProduct.
hybrid search	<code>@search.score</code>	RRF algorithm	Upper limit is bounded by the number of queries being fused, with each query contributing a maximum of approximately $1/k$ to the RRF score (this is the <code>k</code> parameter).

Search method	Parameter	Scoring algorithm	Range
semantic ranking	<code>@search.rerankerScore</code>	Semantic ranking	0.00 - 4.00

Semantic ranking occurs after RRF merging of results. Its score (`@search.rerankerScore`) is always reported separately in the query response. Semantic ranker can rerank full text and hybrid search results, assuming those results include fields having semantically rich content. It can rerank pure vector queries if the search documents include text fields that contain semantically relevant content.

Unpack a search score into subscores

You can deconstruct a search score to view its subscores. For vector queries, this information can help you determine an appropriate value for [vector weighting](#) or [setting minimum thresholds](#).

To get subscores:

- Use the [Search Documents REST API](#) or an Azure SDK package that provides the feature.
- Modify a query request, adding a new `debug` parameter set to either `vector`, `semantic` if using semantic ranker, or `all`.

Here's an example of hybrid query that returns subscores in debug mode:

HTTP

```
POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2025-09-01
```

```
{
  "vectorQueries": [
    {
      "vector": [
        -0.009154141,
        0.018708462,
        ...
        -0.02178128,
        -0.00086512347
      ],
      "fields": "DescriptionVector",
    }
  ]
}
```

```

        "kind": "vector",
        "exhaustive": true,
        "k": 10
    },
    {
        "vector": [
            -0.009154141,
            0.018708462,
            ...
            -0.02178128,
            -0.00086512347
        ],
        "fields": "DescriptionVector",
        "kind": "vector",
        "exhaustive": true,
        "k": 10
    }
],
"search": "historic hotel walk to restaurants and shopping",
"select": "HotelName, Description, Address/City",
"debug": "vector",
"top": 10
}

```

Weighted scores

You can also [weight vector queries](#) to increase or decrease their importance in a hybrid query.

Recall that when computing RRF for a certain document, the search engine looks at the rank of that document for each result set where it shows up. Assume a document shows up in three separate search results, where the results are from two vector queries and one text BM25-ranked query. The position of the document varies in each result.

[] [Expand table](#)

Match found	Position in results	@search.score	weight multiplier	@search.score (weighted)
vector results one	position 1	0.8383955	0.5	0.41919775
vector results two	position 5	0.81514114	2.0	1.63028228
BM25 results	position 10	0.8577363	NA	0.8577363

The document's position in each result set corresponds to an initial score, which is added up to create the final RRF score for that document.

If you add vector weighting, the initial scores are subject to a weighting multiplier that increases or decreases the score. The default is 1.0, which means no weighting and the initial score is used as-is in RRF scoring. However, if you add a weight of 0.5, the score is reduced and that result becomes less important in the combined ranking. Conversely, if you add a weight of 2.0, the score becomes a larger factor in the overall RRF score.

In this example, the `@search.score` (weighted) values are passed to the RRF ranking model.

Number of ranked results in a hybrid query response

By default, if you aren't using pagination, the search engine returns the top 50 highest ranking matches for full text search, and the most similar `k` matches for vector search. In a hybrid query, `top` determines the number of results in the response. Based on defaults, the top 50 highest ranked matches of the unified result set are returned.

Often, the search engine finds more results than `top` and `k`. To return more results, use the paging parameters `top`, `skip`, and `next`. Paging is how you determine the number of results on each logical page and navigate through the full payload. You can [set `maxTextRecallSize`](#) to larger values (the default is 1,000) to return more results from the text side of hybrid query.

By default, full text search is subject to a maximum limit of 1,000 matches (see [API response limits](#)). Once 1,000 matches are found, the search engine no longer looks for more.

For more information, see [How to work with search results](#).

See also

- [Hybrid search](#)
- [Vector search](#)

Add scoring profiles to boost search scores

Scoring profiles are used to boost or suppress the ranking of matching documents based on user-defined criteria. In this article, learn how to specify and assign a scoring profile that boosts a search score based on parameters that you provide. You can create scoring profiles based on:

- Weighted string fields, where boosting is based on a match found in a designated field. For example, matches found in a "Subject" field are considered more relevant than the same match found in a "Description" field.
- Functions for numeric fields, including dates and geographic coordinates. Functions for numeric content support boosting on distance (applies to geographic coordinates), freshness (applies to datetime fields), range, and magnitude.
- Functions for string collections (tags). A tags function boosts a document's search score if any item in the collection is matched by the query.
- (preview) [An aggregation of distinct boosts](#). Within a single scoring profile, you can specify multiple scoring functions, and then set `"functionAggregation": "product"`. Documents that score highly across all functions are prioritized, while those that score weak in one or more fields are suppressed.

You can add a scoring profile to an index by editing its JSON definition in the Azure portal or programmatically through APIs like [Create or Update Index REST](#) or equivalent index update APIs in any Azure SDK. There's no index rebuild requirements so you can add, modify, or delete a scoring profile with no effect on indexed documents.

Prerequisites

- A search index with text or numeric (nonvector) fields.

Rules for scoring profiles

You can use scoring profiles in [keyword search](#), [vector search](#), [hybrid search](#), and with [semantic reranking](#)). However, scoring profiles only apply to nonvector fields, so make sure your index has text or numeric fields that can be boosted or weighted.

You can have up to 100 scoring profiles within an index (see [service Limits](#)), but you can only specify one profile at a time in any given query.

You can use [semantic ranking](#) with scoring profiles and apply a [scoring profile after semantic ranking](#) occurs. Otherwise, when multiple ranking or relevance features are in play, semantic

ranking is the last step. [How search scoring works](#) provides an illustration of the order of operations.

Extra rules apply specifically to functions.

! Note

Unfamiliar with relevance concepts? Visit [Relevance and scoring in Azure AI Search](#) for background. You can also watch this [video segment on YouTube](#) ↗ for scoring profiles over BM25-ranked results.

Scoring profile definition

A scoring profile is defined in an index schema. It consists of weighted fields, functions, and parameters.

The following definition shows a simple profile named "geo". This example boosts results that have the search term in the hotelName field. It also uses the `distance` function to favor results that are within 10 kilometers of the current location. If someone searches on the term 'inn', and 'inn' happens to be part of the hotel name, documents that include hotels with 'inn' within a 10-kilometer radius of the current location appear higher in the search results.

JSON

```
"scoringProfiles": [
  {
    "name": "geo",
    "text": {
      "weights": {
        "hotelName": 5
      }
    },
    "functions": [
      {
        "type": "distance",
        "boost": 5,
        "fieldName": "location",
        "interpolation": "logarithmic",
        "distance": {
          "referencePointParameter": "currentLocation",
          "boostingDistance": 10
        }
      }
    ]
  }
]
```

To use this scoring profile, your query is formulated to specify `scoringProfile` parameter in the request. If you're using the REST API, queries are specified through GET and POST requests. In the following example, "currentLocation" has a delimiter of a single dash (-). It's followed by longitude and latitude coordinates, where longitude is a negative value.

HTTP

```
POST /indexes/hotels/docs&api-version=2025-09-01
{
  "search": "inn",
  "scoringProfile": "geo",
  "scoringParameters": ["currentLocation--122.123,44.77233"]
}
```

Query parameters, including `scoringParameters`, are described in [Search Documents \(REST API\)](#).

For more scenarios, see the examples for [freshness and distance](#) and [weighted text and functions](#) in this article.

Add a scoring profile to a search index

1. Start with an [index definition](#). You can add and update scoring profiles on an existing index without having to rebuild it. Use [Get Index](#) to pull down an existing index, and use [Create or Update Index](#) request to post a revision.
2. Paste in the [template](#) provided in this article.
3. Provide a name that adheres to [naming conventions](#).
4. Specify boosting criteria. A single profile can contain [text weighted fields](#), [functions](#), or both.

You should work iteratively, using a data set that helps you prove or disprove the efficacy of a given profile.

Scoring profiles can be defined in the Azure portal as shown in the following screenshot, or programmatically through [REST APIs](#) or in Azure SDKs, such as the `ScoringProfile` class in [.NET](#) or [Python](#) client libraries.

The screenshot shows the Microsoft Azure portal interface for a search service named 'hotels-sample-index'. At the top, there's a navigation bar with 'Microsoft Azure' and a search bar. Below it, the service name 'hotels-sample-index' is shown with a '... more' button. Underneath, there are tabs for 'Index' (selected), 'Documents' (50 documents, 310.75 kB), 'Storage' (0), and 'Actions' (Save, Discard, Refresh, Create Demo App, Delete). The main content area has tabs for 'Search explorer', 'Fields', 'CORS', 'Scoring profiles' (which is underlined and highlighted with a red box), and 'Index Definition (JSON)'. Below these tabs, there's a section for 'Scoring profiles' with a button '+ Add scoring profile' (also highlighted with a red box) and a checkbox 'Set as default profile'. There are also 'Name' and 'Functions' fields and a search icon. A message at the bottom says 'You haven't created any scoring profiles. Click "Add scoring profile" to create one.'

Template

This section shows the syntax and template for scoring profiles. For a description of properties, see the [REST API reference](#).

JSON

```
"scoringProfiles": [
  {
    "name": "name of scoring profile",
    "text": (optional, only applies to searchable fields) {
      "weights": {
        "searchable_field_name": relative_weight_value (positive #'s),
        ...
      }
    },
    "functions": (optional) [
      {
        "type": "magnitude | freshness | distance | tag",
        "boost": # (positive number used as multiplier for raw score != 1),
        "fieldName": "(...)",
        "interpolation": "constant | linear (default) | quadratic | logarithmic",
        "magnitude": {
          "boostingRangeStart": #,
          "boostingRangeEnd": #,
          "constantBoostBeyondRange": true | false (default)
        }
        // ( - or - )

        "freshness": {
          "boostingDuration": "..." (value representing timespan over which boosting occurs)
        }
        // ( - or - )
      }
    ]
  }
]
```

```

    "distance": {
        "referencePointParameter": "...", (parameter to be passed in queries to
use as reference location)
        "boostingDistance": # (the distance in kilometers from the reference
location where the boosting range ends)
    }

    // ( - or -)

    "tag": {
        "tagsParameter": "..."(parameter to be passed in queries to specify a
list of tags to compare against target field)
    }
},
],
"functionAggregation": (optional, applies only when functions are specified)
"sum (default) | average | minimum | maximum | firstMatching"
}
],
"defaultScoringProfile": (optional) "...",

```

Use text-weighted fields

Use text-weighted fields when field context is important and queries include `searchable` string fields. For example, if a query includes the term "airport", you might favor "airport" in the HotelName field rather than the Description field.

Weighted fields are name-value pairs composed of a `searchable` field and a positive number that is used as a multiplier. If the original field score of HotelName is 3, the boosted score for that field becomes 6, contributing to a higher overall score for the parent document itself.

JSON

```

"scoringProfiles": [
{
    "name": "boostSearchTerms",
    "text": {
        "weights": {
            "HotelName": 2,
            "Description": 5
        }
    }
}
]

```

Use functions

Use functions when simple relative weights are insufficient or don't apply, as is the case of distance and freshness, which are calculations over numeric data. You can specify multiple functions per scoring profile. For more information about the EDM data types used in Azure AI Search, see [Supported data types](#).

[] Expand table

Function	Description	Use cases
distance	Boost by proximity or geographic location. This function can only be used with <code>Edm.GeographyPoint</code> fields.	Use for "find near me" scenarios.
freshness	Boost by values in a datetime field (<code>Edm.DateTimeOffset</code>). Set <code>boostingDuration</code> to specify a value representing a timespan over which boosting occurs.	Use when you want to promote recent (newer) dates. You can also boost items like calendar events with future dates such that are closer to the present, as compared with items that are further into the future. One end of the range is fixed to the current time.
magnitude	Magnitude is the computed distance between the document's value (such as a date or location) and the reference point (such as "now" or a target location). It's the input to the scoring function and determines how much boost is applied. Alter rankings based on the range of values for a numeric field. The value must be an integer or floating-point number. For star ratings of 1 through 4, this would be 1. For margins over 50%, this would be 50. This function can only be used with <code>Edm.Double</code> and <code>Edm.Int</code> fields. For the magnitude function, you can reverse the range, high to low, if you want the inverse pattern (for example, to boost lower-priced items more than higher-priced items). Given a range of prices from \$100 to \$1, you would set <code>boostingRangeStart</code> at 100 and <code>boostingRangeEnd</code> at 1 to boost the lower-priced items.	Use when you want to boost by profit margin, ratings, clickthrough counts, number of downloads, highest price, lowest price, or a count of downloads. When two items are relevant, the item with the higher rating is displayed first.
tag	Boost by tags that are common to both search documents and query strings. Tags are provided in a <code>tagsParameter</code> . This function can only be used with search fields of type <code>Edm.String</code> and <code>Collection(Edm.String)</code> .	Use when you have tag fields. If a given tag within the list is itself a comma-delimited list, you can use a text normalizer on the field to strip out the commas at query time (map the comma character to a space). This approach "flattens" the list

Function	Description	Use cases
		so that all terms are a single, long string of comma-delimited terms.

Freshness and distance scoring are special cases of magnitude-based scoring, where the magnitude is automatically computed from a datetime or geographic field.

Rules for using functions

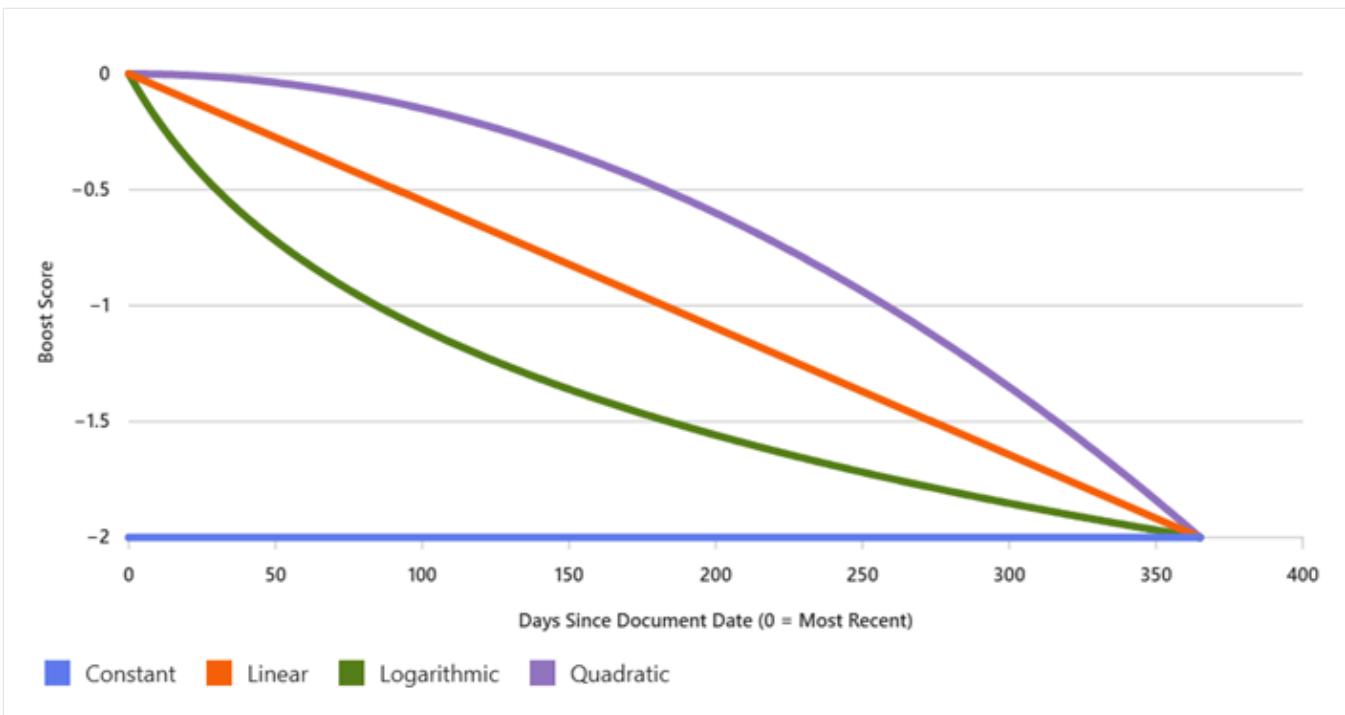
- Functions can only be applied to fields that are attributed as `filterable`.
- Function type ("freshness", "magnitude", "distance", "tag") must be lower case.
- Functions can't include null or empty values.
- Functions can only have a single field per function definition. To use magnitude twice in the same profile, provide two definitions magnitude, one for each field.

Set interpolations

Interpolations set the shape of the slope used for boosting freshness and distance. Because scoring is high to low, the slope is always decreasing, but the interpolation determines the curve of the downward slope and how aggressively the boost score changes as document dates get older.

[] Expand table

Interpolation	Description
<code>linear</code>	For items that are within the max and min range, boosting is applied in a constantly decreasing amount. Recommended when you want a gradual decay in relevance. Linear is the default interpolation for a scoring profile.
<code>constant</code>	For items that are within the start and ending range, a constant boost is applied to the rank results. Use this when you want a flat penalty regardless of age.
<code>quadratic</code>	Quadratic initially decreases at smaller pace and then accelerates as it approaches the end range, it decreases at a much higher interval. Use this interpolation when you want to strongly favor the most recent documents and sharply demote older ones. This interpolation option isn't allowed in the tag scoring function.
<code>logarithmic</code>	Logarithmic initially decreases at higher pace and then as it approaches the end range, it decreases at a much smaller interval. Recommended when you have a strong preference for very recent content but less sensitivity as documents age. This interpolation option isn't allowed in the tag scoring function.



Set boostingDuration for freshness function

`boostingDuration` is an attribute of the `freshness` function. You use it to set an expiration period after which boosting stops for a particular document. For example, to boost a product line or brand for a 10-day promotional period, you would specify the 10-day period as "P10D" for those documents.

`boostingDuration` must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). The pattern for this is: "P[nD][T[nH][nM][nS]]".

The following table provides several examples.

[] [Expand table](#)

Duration	boostingDuration
1 day	"P1D"
2 days and 12 hours	"P2DT12H"
15 minutes	"PT15M"
30 days, 5 hours, 10 minutes, and 6.334 seconds	"P30DT5H10M6.334S"
1 year	"365D"

For more examples, see [XML Schema: Datatypes \(W3.org web site\)](#).

Example: boosting by freshness or distance

The shape of the boost curve (constant, linear, logarithmic, quadratic) affects how aggressively scores change across the range.

When using the freshness function, if you want the boost to have a more dramatic effect on more recent dates, choose a quadratic interpolation. Quadratic amplifies the effect of near recent dates and closer locations and tapers off more slowly at the far end of the range. In contrast, a logarithmic curve shifts more sharply at the far end.

Here's an example scoring profile that demonstrates how to boost by freshness.

JSON

```
{  
  "name": "docs-index",  
  "fields": [  
    { "name": "id", "type": "Edm.String", "key": true, "filterable": true },  
    { "name": "title", "type": "Edm.String", "searchable": true },  
    { "name": "content", "type": "Edm.String", "searchable": true },  
    { "name": "lastUpdated", "type": "Edm.DateTimeOffset", "filterable": true,  
     "sortable": true }  
  ],  
  "scoringProfiles": [  
    {  
      "name": "freshnessBoost",  
      "text": {  
        "weights": {  
          "content": 1.0  
        }  
      },  
      "functions": [  
        {  
          "type": "freshness",  
          "fieldName": "lastUpdated",  
          "boost": 2.0,  
          "interpolation": "quadratic",  
          "parameters": {  
            "boostingDuration": "365D"  
          }  
        }  
      ]  
    }  
  ]  
}
```

- The `freshness` function computes a magnitude from "now" to `lastUpdated`.
- A positive boost with quadratic interpolation increases lift for recent dates, tapering quickly for older ones.

- "boostingDuration": "365D" defines the time window over which freshness is evaluated, for example boosting documents dated within the last year.
- "interpolation": "quadratic" means the boost effect is stronger for documents closer to the current date and tapers off more sharply for older ones.

In the next example, a linear interpolation provides a steady preference for most-recent content across the 30-day window. Increase boost if the signal needs to win against other relevance factors.

JSON

```
{
  "name": "freshness30_linear",
  "functions": [
    {
      "type": "freshness",
      "fieldName": "lastUpdated",
      "boost": 3.0,
      "interpolation": "linear",
      "parameters": { "boostingDuration": "P30D" }
    }
  ]
}
```

Example: boosting by weighted text and functions

Tip

See this [blog post](#) and [notebook](#) for a demonstration of using scoring profiles and document boosting in vector and generative AI scenarios.

The following example shows the schema of an index with two scoring profiles (`boostGenre`, `newAndHighlyRated`). Any query against this index that includes either profile as a query parameter uses the profile to score the result set.

The `boostGenre` profile uses weighted text fields, boosting matches found in `albumTitle`, `genre`, and `artistName` fields. The fields are boosted 1.5, 5, and 2 respectively. Why is `genre` boosted so much higher than the others? If search is conducted over data that is somewhat homogeneous (as is the case with '`genre`' in the `musicstoreindex`), you might need a larger variance in the relative weights. For example, in the `musicstoreindex`, '`rock`' appears as both a genre and in identically phrased genre descriptions. If you want `genre` to outweigh `genre` description, the `genre` field needs a much higher relative weight.

JSON

```
{
  "name": "musicstoreindex",
  "fields": [
    { "name": "key", "type": "Edm.String", "key": true },
    { "name": "albumTitle", "type": "Edm.String" },
    { "name": "albumUrl", "type": "Edm.String", "filterable": false },
    { "name": "genre", "type": "Edm.String" },
    { "name": "genreDescription", "type": "Edm.String", "filterable": false },
    { "name": "artistName", "type": "Edm.String" },
    { "name": "orderableOnline", "type": "Edm.Boolean" },
    { "name": "rating", "type": "Edm.Int32" },
    { "name": "tags", "type": "Collection(Edm.String)" },
    { "name": "price", "type": "Edm.Double", "filterable": false },
    { "name": "margin", "type": "Edm.Int32", "retrievable": false },
    { "name": "inventory", "type": "Edm.Int32" },
    { "name": "lastUpdated", "type": "Edm.DateTimeOffset" }
  ],
  "scoringProfiles": [
    {
      "name": "boostGenre",
      "text": {
        "weights": {
          "albumTitle": 1.5,
          "genre": 5,
          "artistName": 2
        }
      }
    },
    {
      "name": "newAndHighlyRated",
      "functions": [
        {
          "type": "freshness",
          "fieldName": "lastUpdated",
          "boost": -10,
          "interpolation": "quadratic",
          "freshness": {
            "boostingDuration": "P365D"
          }
        },
        {
          "type": "magnitude",
          "fieldName": "rating",
          "boost": 10,
          "interpolation": "linear",
          "magnitude": {
            "boostingRangeStart": 1,
            "boostingRangeEnd": 5,
            "constantBoostBeyondRange": false
          }
        }
      ]
    }
  ]
}
```

```
]  
}
```

Example: function aggregation

! Note

This capability is currently in preview, available through the [2025-11-01-preview REST API](#) and in Azure SDK preview packages that provide the feature.

Within a single scoring profile, you can specify multiple scoring functions, and then set `"functionAggregation": "product"`. Documents that score highly across all functions are prioritized, while those that score weak in one or more fields are suppressed.

In this example, create a scoring profile that includes two boosting functions that boost by `rating` and `baseRate`, and then set `functionAggregation` to `product`.

HTTP

```
### Create a new index  
PUT {{url}}/indexes/hotels-scoring?api-version=2025-11-01-preview  
Content-Type: application/json  
api-key: {{key}}  
  
{  
    "name": "hotels-scoring",  
    "fields": [  
        {"name": "HotelId", "type": "Edm.String", "key": true, "filterable": true,  
        "facetable": true},  
        {"name": "HotelName", "type": "Edm.String", "searchable": true,  
        "filterable": false, "sortable": true, "facetable": true},  
        {"name": "Description", "type": "Edm.String", "searchable": true,  
        "filterable": false, "sortable": false, "facetable": true, "analyzer": "en.lucene"},  
        {"name": "Category", "type": "Edm.String", "searchable": true, "filterable":  
        true, "sortable": true, "facetable": true},  
        {"name": "Tags", "type": "Collection(Edm.String)", "searchable": true,  
        "filterable": true, "sortable": false, "facetable": true},  
        {"name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true,  
        "sortable": true, "facetable": true},  
        {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset", "filterable":  
        true, "sortable": true, "facetable": true},  
        {"name": "Rating", "type": "Edm.Double", "filterable": true, "sortable":  
        true, "facetable": true},  
        {"name": "BaseRate", "type": "Edm.Double", "filterable": true, "sortable":  
        true, "facetable": true },  
        {"name": "Address", "type": "Edm.ComplexType",  
        "fields": [  
            {"name": "StreetAddress", "type": "Edm.String", "filterable": false,
```

```

"sortable": false, "facetable": true, "searchable": true},
        {"name": "City", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "StateProvince", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "PostalCode", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true}
    ]
},
],
"scoringProfiles": [
{
    "name": "productAggregationProfile",
    "functions": [
        {
            "type": "magnitude",
            "fieldName": "Rating",
            "boost": 2.0,
            "interpolation": "linear",
            "magnitude": {
                "boostingRangeStart": 1.0,
                "boostingRangeEnd": 5.0,
                "constantBoostBeyondRange": false
            }
        },
        {
            "type": "magnitude",
            "fieldName": "BaseRate",
            "boost": 1.5,
            "interpolation": "linear",
            "magnitude": {
                "boostingRangeStart": 50.0,
                "boostingRangeEnd": 400.0,
                "constantBoostBeyondRange": false
            }
        }
    ],
    "functionAggregation": "product"
},
],
"defaultScoringProfile": "productAggregationProfile"
}

```

This next request loads the index with searchable content that tests the profile.

HTTP

```

### Upload documents to the index
POST {{url}}/indexes/hotels-scoring/docs/index?api-version=2025-11-01-preview
Content-Type: application/json
api-key: {{key}}


{
    "value": [

```

```
{
  "@search.action": "upload",
  "HotelId": "1",
  "HotelName": "Stay-Kay City Hotel",
  "Description": "This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.",
  "Category": "Boutique",
  "Tags": [ "view", "air conditioning", "concierge" ],
  "ParkingIncluded": false,
  "LastRenovationDate": "2022-01-18T00:00:00Z",
  "Rating": 3.60,
  "BaseRate": 200.0,
  "Address":
  {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY",
    "PostalCode": "10022"
  }
},
{
  "@search.action": "upload",
  "HotelId": "2",
  "HotelName": "Old Century Hotel",
  "Description": "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.",
  "Category": "Boutique",
  "Tags": [ "pool", "free wifi", "concierge" ],
  "ParkingIncluded": false,
  "LastRenovationDate": "2019-02-18T00:00:00Z",
  "Rating": 3.60,
  "BaseRate": 150.0,
  "Address":
  {
    "StreetAddress": "140 University Town Center Dr",
    "City": "Sarasota",
    "StateProvince": "FL",
    "PostalCode": "34243"
  }
},
{
  "@search.action": "upload",
  "HotelId": "3",
  "HotelName": "Gastronomic Landscape Hotel",
  "Description": "The Gastronomic Landscape Hotel stands out for its culinary excellence under the management of William Dough, who advises on and oversees all of the Hotel's restaurant services.",
  "Category": "Suite",
  "Tags": [ "restaurant", "bar", "continental breakfast" ],
}
```

```

    "ParkingIncluded": true,
    "LastRenovationDate": "2015-09-20T00:00:00Z",
    "Rating": 4.80,
    "BaseRate": 350.0,
    "Address":
    {
        "StreetAddress": "3393 Peachtree Rd",
        "City": "Atlanta",
        "StateProvince": "GA",
        "PostalCode": "30326"
    }
},
{
    "@search.action": "upload",
    "HotelId": "4",
    "HotelName": "Sublime Palace Hotel",
    "Description": "Sublime Palace Hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking distance to the sites and landmarks of the city and is surrounded by the extraordinary beauty of churches, buildings, shops and monuments. Sublime Cliff is part of a lovingly restored 19th century resort, updated for every modern convenience.",
    "Tags": [ "concierge", "view", "air conditioning" ],
    "ParkingIncluded": true,
    "LastRenovationDate": "2020-02-06T00:00:00Z",
    "Rating": 4.60,
    "BaseRate": 275.0,
    "Address":
    {
        "StreetAddress": "7400 San Pedro Ave",
        "City": "San Antonio",
        "StateProvince": "TX",
        "PostalCode": "78216"
    }
}
]
}

```

Run a query that uses the criteria in the scoring profile to boost results based on a high rating and high base rate. The boosting scores are aggregated to further promote results that score high in both functions.

HTTP

```

### Search with boost
POST {{url}}/indexes/hotels-scoring/docs/search?api-version=2025-11-01-preview
Content-Type: application/json
api-key: {{key}}


{
    "search": "expensive and good hotels",
    "count": true,
    "select": "HotelId, HotelName, Description, Rating, BaseRate",

```

```
        "scoringProfile": "productAggregationProfile"
    }
```

The top response for this query is "Gastronomic Landscape Hotel" with a search score that is almost twice as high as next closest match. This particular hotel has both the highest rating and the highest base rate, so the compounding of both functions promotes this match to the top.

JSON

```
{
    "@odata.count": 4,
    "value": [
        {
            "@search.score": 1.0541908,
            "HotelId": "3",
            "HotelName": "Gastronomic Landscape Hotel",
            "Description": "The Gastronomic Hotel stands out for its culinary excellence under the management of William Dough, who advises on and oversees all of the Hotel\u2019s restaurant services.",
            "Rating": 4.8,
            "BaseRate": 350.0
        },
        {
            "@search.score": 0.53451097,
            "HotelId": "2",
            "HotelName": "Old Century Hotel",
            "Description": "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts. The hotel also regularly hosts events like wine tastings, beer dinners, and live music.",
            "Rating": 3.6,
            "BaseRate": 150.0
        },
        {
            "@search.score": 0.53185254,
            "HotelId": "1",
            "HotelName": "Stay-Kay City Hotel",
            "Description": "This classic hotel is fully-refurbished and ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Times Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.",
            "Rating": 3.6,
            "BaseRate": 200.0
        },
        {
            "@search.score": 0.44853577,
            "HotelId": "4",
            "HotelName": "Sublime Palace Hotel",
            "Description": "Sublime Palace Hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking distance to the sites and landmarks of the city and is surrounded by the
        }
    ]
}
```

```
extraordinary beauty of churches, buildings, shops and monuments. Sublime Cliff is
part of a lovingly restored 19th century resort, updated for every modern
convenience.",  
        "Rating": 4.6,  
        "BaseRate": 275.0  
    }  
}  
}
```

Tuning tips

- Start conservative: boost in the 1.25–2.0 range; increase only if recency is truly decisive.
- Window sizing: Use P30D for hot content, P90D/P180D for moderate recency, P365D for long-tail.
- Interpolation choice:
 - quadratic when you want a strong push to recent.
 - linear when you want a steady gradient.
 - logarithmic when you want a gentle preference.
- Aggregation: If combining multiple functions, sum is easiest; switch to max when you want a single signal to dominate

Last updated on 11/23/2025

Semantic ranking in Azure AI Search

In Azure AI Search, *semantic ranker* is a feature that measurably improves search relevance by using Microsoft's language understanding models to rerank search results. Semantic ranker is also built into [agentic retrieval](#). This article is a high-level introduction to help you understand the behaviors and benefits of semantic ranker.

Semantic ranker is a premium feature, billed by usage, but you can use it for free subject to [service limits](#) for the free tier. We recommend this article for background, but if you'd rather get started, [follow these steps](#).

What is semantic ranking?

Semantic ranker is a collection of query-side capabilities that improve the quality of an initial [BM25-ranked](#) or [RRF-ranked](#) search result for text-based queries, the text portion of vector queries, and hybrid queries. Semantic ranking extends the query execution pipeline in three ways:

- First, it always adds secondary ranking over an initial result set that was scored using BM25 or Reciprocal Rank Fusion (RRF). This secondary ranking uses multi-lingual, deep learning models adapted from Microsoft Bing to promote the most semantically relevant results.
- Second, it returns captions and optionally extracts answers in the response, which you can render on a search page to improve the user's search experience.
- Third, if you enable query rewrite, it expands an initial query string into multiple semantically similar query strings.

Secondary ranking and "answers" apply to the query response. Query rewrite is part of the query request.

Here are the capabilities of the semantic reranker.

 Expand table

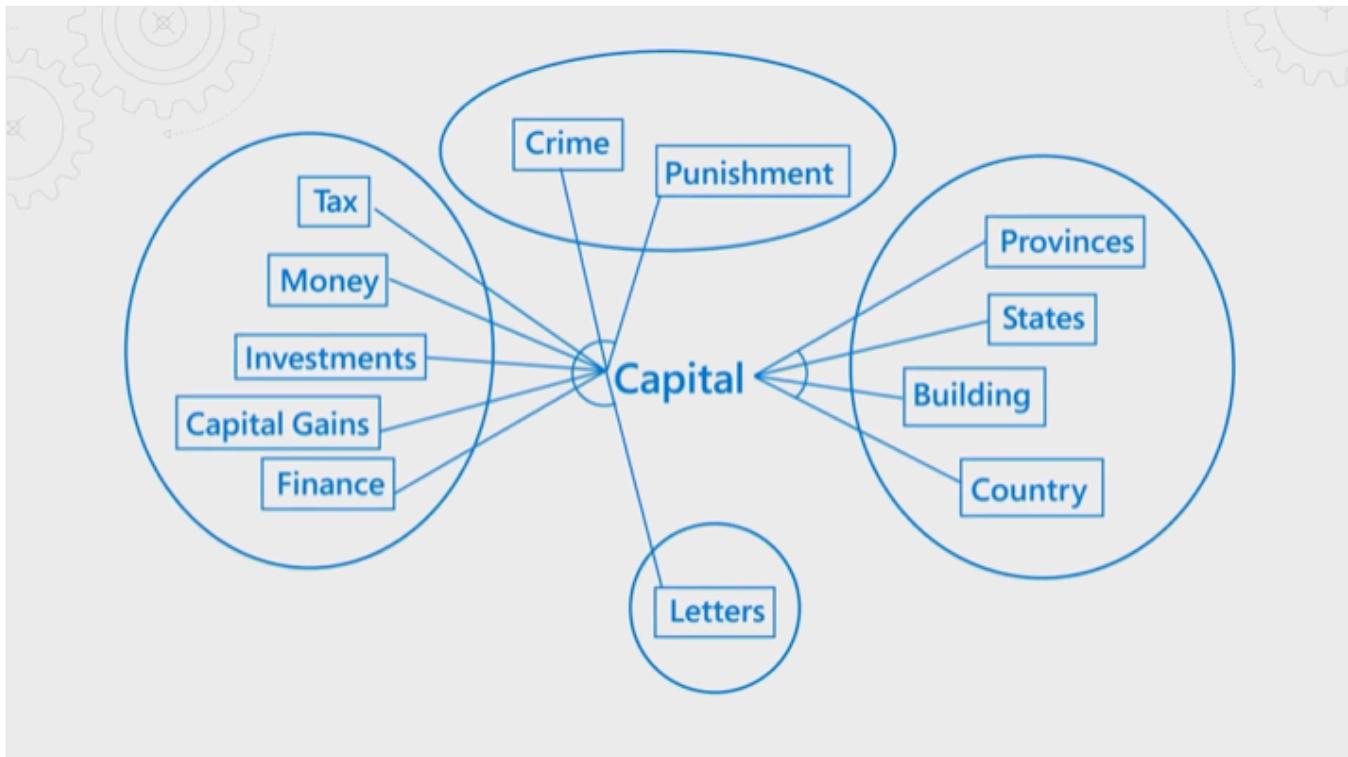
Capability	Description
L2 ranking	Uses the context or semantic meaning of a query to compute a new relevance score over preranked results.
Semantic captions and highlights	Extracts verbatim sentences and phrases from fields that best summarize the content, with highlights over key passages for easy scanning. Captions that summarize a result are useful when individual content fields are too dense for the search results page.

Capability	Description
	Highlighted text elevates the most relevant terms and phrases so that users can quickly determine why a match was considered relevant.
Semantic answers	An optional and extra substructure returned from a semantic query. It provides a direct answer to a query that looks like a question. It requires that a document has text with the characteristics of an answer.
Query rewrite	Using text queries or the text portion of a vector query, semantic ranker creates up to 10 variants of the query, perhaps correcting typos or spelling errors, or rephrasing a query using generated synonyms. The rewritten query runs on the search engine. The results are scored using BM25 or RRF scoring, and then rescored by semantic ranker.

How semantic ranker works

Semantic ranker feeds a query and results to language understanding models hosted by Microsoft and scans for better matches.

The following illustration explains the concept. Consider the term "capital". It has different meanings depending on whether the context is finance, law, geography, or grammar. Through language understanding, the semantic ranker can detect context and promote results that fit query intent.



Semantic ranking is both resource and time intensive. In order to complete processing within the expected latency of a query operation, inputs to the semantic ranker are consolidated and reduced so that the reranking step can be completed as quickly as possible.

There are three steps to semantic ranking:

- Collect and summarize inputs
- Score results using the semantic ranker
- Output rescored results, captions, and answers

How inputs are collected and summarized

In semantic ranking, the query subsystem passes search results as an input to summarization and ranking models. Because the ranking models have input size constraints and are processing intensive, search results must be sized and structured (summarized) for efficient handling.

1. Semantic ranker starts with a [BM25-ranked result](#) from a text query or an [RRF-ranked result](#) from a vector or hybrid query. Only text is used in the reranking exercise, and only the top 50 results progress to semantic ranking, even if results include more than 50. Typically, fields used in semantic ranking are informational and descriptive.
2. For each document in the search result, the summarization model accepts up to 2,000 tokens, where a token is approximately 10 characters. Inputs are assembled from the "title", "keyword", and "content" fields listed in the [semantic configuration](#).
3. Excessively long strings are trimmed to ensure the overall length meets the input requirements of the summarization step. This trimming exercise is why it's important to add fields to your semantic configuration in priority order. If you have very large documents with text-heavy fields, anything after the maximum limit is ignored.

[] [Expand table](#)

Semantic field	Token limit
"title"	128 tokens
"keywords"	128 tokens
"content"	remaining tokens

4. Summarization output is a summary string for each document, composed of the most relevant information from each field. Summary strings are sent to the ranker for scoring, and to machine reading comprehension models for captions and answers.

As of November 2024, the maximum length of each generated summary string passed to the semantic ranker is 2,048 tokens. Previously, it was 256 tokens.

How results are scored

Scoring is done over the caption, and any other content from the summary string that fills out the 2,048 token length.

1. Captions are evaluated for conceptual and semantic relevance, relative to the query provided.
2. A `@search.rerankerScore` is assigned to each document based on the semantic relevance of the document for the given query. Scores range from 4 to 0 (high to low), where a higher score indicates higher relevance.

[] [Expand table](#)

Score	Meaning
4.0	The document is highly relevant and answers the question completely, though the passage might contain extra text unrelated to the question.
3.0	The document is relevant but lacks details that would make it complete.
2.0	The document is somewhat relevant; it answers the question either partially or only addresses some aspects of the question.
1.0	The document is related to the question, and it answers a small part of it.
0.0	The document is irrelevant.

3. Matches are listed in descending order by score and included in the query response payload. The payload includes answers, plain text and highlighted captions, and any fields that you marked as retrievable or specified in a select clause.

! Note

For any given query, the distributions of `@search.rerankerScore` can exhibit slight variations due to conditions at the infrastructure level. Ranking model updates have also been known to affect the distribution. For these reasons, if you're writing custom code for minimum thresholds, or [setting the threshold property](#) for vector and hybrid queries, don't make the limits too granular.

Outputs of semantic ranker

From each summary string, the machine reading comprehension models find passages that are the most representative.

Outputs are:

- A [semantic caption](#) for the document. Each caption is available in a plain text version and a highlight version, and is frequently fewer than 200 words per document.
- An optional [semantic answer](#), assuming you specified the `answers` parameter, the query was posed as a question, and a passage is found in the long string that provides a likely answer to the question.

Captions and answers are always verbatim text from your index. There's no generative AI model in this workflow that creates or composes new content.

Semantic capabilities and limitations

What semantic ranker *can* do:

- Promote matches that are semantically closer to the intent of original query.
- Find strings to use as captions and answers. Captions and answers are returned in the response and can be rendered on a search results page.

What semantic ranker *can't* do is rerun the query over the entire corpus to find semantically relevant results. Semantic ranking reranks the existing result set, consisting of the top 50 results as scored by the default ranking algorithm. Furthermore, semantic ranker can't create new information or strings. Captions and answers are extracted verbatim from your content so if the results don't include answer-like text, the language models won't produce one.

Although semantic ranking isn't beneficial in every scenario, certain content can benefit significantly from its capabilities. The language models in semantic ranker work best on searchable content that is information-rich and structured as prose. A knowledge base, online documentation, or documents that contain descriptive content see the most gains from semantic ranker capabilities.

The underlying technology is from Bing and Microsoft Research, and integrated into the Azure AI Search infrastructure as an add-on feature. For more information about the research and AI investments backing semantic ranker, see [How AI from Bing is powering Azure AI Search \(Microsoft Research Blog\)](#).

The following video provides an overview of the capabilities.

https://www.youtube-nocookie.com/embed/yOf0WfVd_V0

How semantic ranker uses synonym maps

If you have already enabled support for [synonym maps associated to a field](#) in your search index, and that field is included in the [semantic ranker configuration](#), the semantic ranker will automatically apply the configured synonyms during the reranking process.

Availability and pricing

Semantic ranker is available [in selected regions](#). It's used as a standalone feature and as a built-in component of [agentic retrieval](#).

You can disable semantic ranker for your search service, use it on a limited basis for free, or use it more expansively with pay-as-you-go billing:

 [Expand table](#)

Plan	Description
Free	A free tier search service provides 1,000 semantic ranker requests per month and 50 million free agentic reasoning tokens per month. Higher tiers can also use the free plan.
Standard	The standard plan is pay-as-you-go pricing once the monthly free quota is consumed. After the first 1,000 semantic ranker requests, you are charged for each additional 1,000 requests. After the first 50 million agentic reasoning tokens per month, you are charged a nominal fee for each one million agentic reasoning tokens. The transition from Free to Standard is seamless. You aren't notified when the transition occurs. For more information about charges by currency, see the Azure AI Search pricing page .

The [Azure AI Search pricing page](#) shows you the billing rate for different currencies and intervals.

Charges for semantic ranker are levied when query requests include `queryType=semantic` and the search string isn't empty (for example, `search=pet friendly hotels in New York`). If your search string is empty (`search=*`), you aren't charged, even if the `queryType` is set to `semantic`.

How to get started with semantic ranker

1. [Check regional availability](#).
2. [Sign in to Azure portal](#).
3. [Configure semantic ranker for the search service, choosing a pricing plan](#). The free plan is the default.
4. [Configure semantic ranker in a search index](#).

5. Set up queries to return semantic captions and highlights.

6. Optionally, return semantic answers.

See also

- [Blog: Outperforming vector search with hybrid retrieval and ranking capabilities ↗](#)
-

Last updated on 11/19/2025

Enable or disable semantic ranker

Semantic ranker is a premium feature billed by usage. By default, semantic ranker is enabled on a new billable search service and it's configured for the free plan, but anyone with *Contributor* permissions can disable it or change the billing plan. If you don't want anyone to use the feature, you can [disable it service-wide using the management REST API](#). If you disable semantic ranking, you also disable [agentic retrieval](#).

Check availability

To check if semantic ranker is available in your region, see the [Azure AI Search regions list](#).

Enable semantic ranker

Semantic ranker might not be enabled on older services. Follow these steps to enable [semantic ranker](#) at the service level. Once enabled, it's available to all indexes. You can't turn it on or off for specific indexes.

Azure portal

1. Open the [Azure portal](#).
2. Navigate to your search service. On the **Overview** page, make sure the pricing tier is set to **Basic** or higher.
3. On the left-navigation pane, select **Settings > Premium features**.
4. Select either the **Free plan** (default) or the **Standard plan**. You can switch between the free plan and the standard plan at any time.

The screenshot shows the 'Premium features' section of the Azure AI Search service. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Search management, Settings, Premium features (which is highlighted with a red box), Knowledge Center, Keys, Scale, and Search traffic analytics. The main area has a heading 'Premium features' with a sub-section 'Availability'. It shows two plans: 'Free' and 'Standard'. The 'Free' plan is highlighted with a red box and includes details: 1,000 semantic ranker requests per month and 50M free agentic retrieval tokens per month. It costs '\$0.00/month' and has a 'Selected Plan' button. The 'Standard' plan includes details: First 1,000 semantic ranker requests per month free, \$1.00 per additional 1,000 requests; First 50M agentic retrieval tokens per month free, \$0.022 per additional 1M agentic retrieval tokens. It has a 'Select Plan' button.

The free plan is capped at 1,000 queries per month. After the first 1,000 queries in the free plan, an error message indicates you exhausted your quota on the next semantic query. When quota is exhausted, you can upgrade to the standard plan to continue using semantic ranking.

Disable semantic ranker using the REST API

To turn off feature enablement, or for full protection against accidental usage and charges, you can disable semantic ranker by using the [Create or Update Service API](#) on your search service. After the feature is disabled, any requests that include the semantic query type are rejected.

Management REST API calls are authenticated through Microsoft Entra ID. For instructions on how to authenticate, see [Manage your Azure AI Search service with REST APIs](#).

HTTP

PATCH

```
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-service-name}}?api-version=2025-05-01
{
  "properties": {
    "semanticSearch": "disabled"
  }
}
```

To re-enable semantic ranker, run the previous request again and set `semanticSearch` to either **Free** (default) or **Standard**.

Next step

[Configure semantic ranker](#)

Last updated on 11/18/2025

Configure semantic ranker and return captions in search results

Semantic ranking iterates over an initial result set, applying an L2 ranking methodology that promotes the most semantically relevant results to the top of the stack. You can also get semantic captions, with highlights over the most relevant terms and phrases, and [semantic answers](#).

This article explains how to configure a search index for semantic reranking.

(!) Note

If you have existing code that calls preview or previous API versions, see [Migrate semantic ranking code](#) for help with modifying your code.

Prerequisites

- Azure AI Search in any [region that provides semantic ranking](#).
- Semantic ranker [enabled on your search service](#).
- An existing search index with rich text content. Semantic ranking applies to strings (nonvector) fields and works best on content that is informational or descriptive.

Choose a client

You can specify a semantic configuration on new or existing indexes, using any of the following tools and software development kits (SDKs) to add a semantic configuration:

- [Azure portal](#), using the index designer to add a semantic configuration.
- [Visual Studio Code](#) with the [REST client](#) and a [Create or Update Index \(REST\) API](#).
- [Azure SDK for .NET](#)
- [Azure SDK for Python](#)
- [Azure SDK for Java](#)
- [Azure SDK for JavaScript](#)

Add a semantic configuration

Some workloads create a semantic configuration automatically. If you're using [agentic retrieval](#) and a [knowledge source that indexes content](#) on Azure AI Search, your generated index

already has a semantic configuration that works for your content.

For other workloads, you can set up a semantic configuration yourself. A *semantic configuration* is a section in your index that establishes the field inputs used for semantic ranking. You can add or update a semantic configuration at any time, no rebuild necessary. If you create multiple configurations, you can specify a default. At query time, specify a semantic configuration on a [query request](#), or leave it blank to use the default.

You can create up to 100 semantic configurations in a single index.

A semantic configuration has a name and the following properties:

 [Expand table](#)

Property	Characteristics
Title field	A short string, ideally under 25 words. This field could be the title of a document, name of a product, or a unique identifier. If you don't have suitable field, leave it blank.
Content fields	Longer chunks of text in natural language form, subject to maximum token input limits on the machine learning models. Common examples include the body of a document, description of a product, or other free-form text.
Keyword fields	A list of keywords, such as the tags on a document, or a descriptive term, such as the category of an item.

You can only specify one title field, but you can have as many content and keyword fields as you like. For content and keyword fields, list the fields in priority order because lower priority fields might get truncated.

Across all semantic configuration properties, the fields you assign must be:

- Attributed as `searchable` and `retrievable`
- Strings of type `Edm.String`, `Collection(Edm.String)`, string subfields of `Edm.ComplexType`

Azure portal

1. Sign in to the [Azure portal](#)  and navigate to a search service that has [semantic ranking enabled](#).
2. From **Indexes** on the left-navigation pane, select an index.
3. Select **Semantic configurations** and then select **Add semantic configuration**.

Microsoft Azure

All services > search-service-contoso-search-service-centralus | Overview > contoso-search-service-centralus | Indexes >

hotels-sample-index

Save Discard Refresh Create demo app Edit JSON Delete Encryption

Documents Total storage Vector index size Max storage

50 560.69 KB 0 Bytes 15 GB

Search explorer Fields CORS Scoring profiles Semantic configurations Vector profiles

+ Add semantic configuration Delete

You haven't created any semantic configurations

Create

This screenshot shows the Microsoft Azure portal interface for a search service named 'contoso-search-service-centralus'. Under the 'Indexes' section, the 'hotels-sample-index' is selected. The 'Semantic configurations' tab is active, highlighted with a red box. Below it, a button '+ Add semantic configuration' is also highlighted with a red box. The main content area displays a message 'You haven't created any semantic configurations' with a 'Create' button below it. A large hexagonal icon is centered in the background.

4. On the **New semantic configuration** page, enter a semantic configuration name and select the fields to use in the semantic configuration. Only searchable and retrievable string fields are eligible. Make sure to list content fields and keyword fields in priority order.

hotels-sample-index

Save Discard Refresh Create demo app Edit JSON Delete

Documents Total storage Vector index size Max storage

50 560.69 KB 0 Bytes 15 GB

Search explorer Fields CORS Scoring profiles Semantic configurations

+ Add semantic configuration Delete

You haven't created any semantic configurations

Create

New semantic configuration

Name * my-semantic-config

Title field HotelName

Content fields

Field name Description Please select a field

Keyword fields

Field name Tags

Save Cancel

This screenshot shows the 'New semantic configuration' dialog box. It has a red box around the 'Name' input field which contains 'my-semantic-config'. Below it, the 'Title field' dropdown is set to 'HotelName'. Under 'Content fields', there is a 'Field name' dropdown with 'Description' and a second dropdown below it with 'Please select a field'. Under 'Keyword fields', there is a 'Field name' dropdown with 'Tags'. At the bottom right of the dialog, there is a 'Save' button highlighted with a red box and a 'Cancel' button.

5. Select **Save** to save the configuration settings.
6. Select **Save** again on the index page to save the semantic configuration in the index.

Opt in for prerelease semantic ranking models

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Using [previewREST APIs](#) and preview Azure SDKs that provide the property, you can optionally configure an index to use prerelease semantic ranking models if one is deployed in your region. There's no mechanism for knowing if a prerelease is available, or if it was used on specific query. For this reason, we recommend that you use this property in test environments, and only if you're interested in trying out the very latest semantic ranking models.

The configuration property is `"flightingOptIn": true`, and it's set in the semantic configuration section of an index. The property is null or false by default. You can set it true on a create or update request at any time, and it affects semantic queries moving forward, assuming the query stipulates a semantic configuration that includes the property.

rest

```
PUT https://myservice.search.windows.net/indexes('hotels')?  
allowIndexDowntime=False&api-version=2025-11-01-preview
```

```
{  
    "name": "hotels",  
    "fields": [ ],  
    "scoringProfiles": [ ],  
    "defaultScoringProfile": "geo",  
    "suggesters": [ ],  
    "analyzers": [ ],  
    "corsOptions": { },  
    "encryptionKey": { },  
    "similarity": { },  
    "semantic": {  
        "configurations": [  
            {  
                "name": "semanticHotels",  
                "prioritizedFields": {  
                    "titleField": {  
                        "fieldName": "hotelName"  
                    },  
                    "prioritizedContentFields": [  
                        {  
                            "fieldName": "description"  
                        },  
                        {  
                            "fieldName": "description_fr"  
                        }  
                    ]  
                }  
            }  
        ]  
    }  
}
```

```
],
  "prioritizedKeywordsFields": [
    {
      "fieldName": "tags"
    },
    {
      "fieldName": "category"
    }
  ],
  "flightingOptIn": true
}
},
"vectorSearch": { }
}
```

Next steps

Test your semantic configuration by running a semantic query.

[Create a semantic query](#)

Last updated on 11/21/2025

Add semantic ranking to queries in Azure AI Search

You can apply semantic ranking to text queries, hybrid queries, and vector queries if your search documents contain string fields and the [vector query has a text representation](#) in the search document.

This article explains how to invoke the semantic ranker on queries. It assumes you're using the most recent stable or preview APIs. For help with older versions, see [Migrate semantic ranking code](#).

Prerequisites

- Azure AI Search in any [region that provides semantic ranking](#), with [semantic ranker enabled](#).
- An existing search index with a [semantic configuration](#) and rich text content.
- Familiarity with [semantic ranking](#).

Note

Captions and answers are extracted verbatim from text in the search document. The semantic subsystem uses machine reading comprehension to recognize content having the characteristics of a caption or answer, but doesn't compose new sentences or phrases except in the case of [query rewrite](#). For this reason, content that includes explanations or definitions work best for semantic ranking. If you want chat-style interaction with generated responses, see [Agentic retrieval](#) or [Retrieval Augmented Generation \(RAG\)](#).

Choose a client

You can use any of the following tools and SDKs to build a query that uses semantic ranking:

- [Azure portal](#) ↗, using the index designer to add a semantic configuration.
- [Visual Studio Code](#) ↗ with a [REST client](#) ↗
- [Azure SDK for .NET](#) ↗
- [Azure SDK for Python](#) ↗
- [Azure SDK for Java](#) ↗
- [Azure SDK for JavaScript](#) ↗

Avoid features that bypass relevance scoring

A few query capabilities bypass relevance scoring, which makes them incompatible with semantic ranking. If your query logic includes the following features, you can't semantically rank your results:

- A query with `search=*` or an empty search string, such as pure filter-only query, won't work because there's nothing to measure semantic relevance against and so the search scores are zero. The query must provide terms or phrases that can be evaluated during processing, and that produces search documents that are scored for relevance. Scored results are inputs to the semantic ranker.
- Sorting (`orderBy` clauses) on specific fields overrides search scores and a semantic score. Given that the semantic score is supposed to provide the ranking, adding an `orderBy` clause results in an HTTP 400 error if you apply semantic ranking over ordered results.

Set up the query

By default, queries don't use semantic ranking. To use semantic ranking, two different parameters can be used. Each parameter supports a different set of query formats.

All semantic queries, whether specified through `search` plus `queryType`, or through `semanticQuery`, must be plain text and they can't be empty. As you can see from the table below, the `queryType-semantic` parameter supports a subset of query formats.

[+] Expand table

Parameter	Plain text search	Simple text search syntax	Full text search syntax	Vector search	Hybrid Search	Semantic answers and captions
<code>queryType-semantic</code> ¹	✓	✗	✗	✗	✓	✓
<code>semanticQuery="<your plain text query>"</code> ²	✓	✓	✓	✓	✓	✓

¹ `queryType=semantic` can't support explicit `simple` or `full` values because the `queryType` parameter is being used for `semantic`. The effective query behaviors are the defaults of the simple parser.

² The `semanticQuery` parameter can be used for all query types. However, it isn't supported in the Azure portal [Search Explorer](#).

Regardless of the parameter chosen, the index should contain text fields with rich semantic content and a [semantic configuration](#).

Azure portal

Search explorer includes options for semantic ranking. Recall that you can't set the `semanticQuery` parameter in the Azure portal.

1. Sign in to the [Azure portal](#).
2. Open a search index and select **Search explorer**.
3. Select **Query options**. If you already defined a semantic configuration, it's selected by default. If you don't have one, [create a semantic configuration](#) for your index.
4. Enter a query, such as "historic hotel with good food", and select **Search**.
5. Alternatively, select **JSON view** and paste definitions into the query editor. The Azure portal doesn't support using `semanticQuery`, so setting `queryType` to "`semantic`" is required:

The screenshot shows the Azure portal interface for a search index named "hotels-sample-index". The "Search explorer" tab is selected. In the top right, there are dropdown menus for "2023-10-01-Preview" and "View". A red box highlights the "JSON query editor" section, which contains the following JSON code:

```
1 {  
2   "search": "historic hotel with good food",  
3   "select": "HotelName, Description, Category, Tags",  
4   "queryType": "semantic",  
5   "semanticConfiguration": "my-semantic-config",  
6   "captions": "extractive",  
7   "answers": "extractive|count-3",  
8   "queryLanguage": "en-US"  
9 }
```

Next to the JSON editor, a red box highlights the "JSON view" button in the "View" dropdown menu. Below the editor, the "Results" section displays the search results in JSON format. One result is shown in detail:

```
1 {  
2   "@odata.context":  
3   "@search.answers": [],  
4   "value": [  
5     {  
6       "@search.score": 1.4323225,  
7       "@search.rerankerScore": 2.344247817993164,  
8       "@search.captions": [  
9         {  
10           "text": "Nordick's Motel. Only 90 miles (about 2 hours) from the nation's capital and nearby mos",  
11           "highlights": ""  
12         }  
13       ],  
14       "HotelName": "Nordick's Motel",  
15       "Description": "Only 90 miles (about 2 hours) from the nation's capital and nearby most everything t",  
16       "Category": "Resort and Spa",  
17       "Tags": [  
18         "restaurant",  
19         "air conditioning",  
20         "restaurant"  
21       ]  
22     },
```

On the right side of the results panel, there is a vertical scrollable list of other search results.

JSON example for setting query type to semantic that you can paste into the view:

JSON

```
{  
  "search": "funky or interesting hotel with good food on site",  
  "count": true,  
  "queryType": "semantic",  
  "semanticConfiguration": "my-semantic-config",  
  "captions": "extractive|highlight-true",  
  "answers": "extractive|count-3",  
  "highlightPreTag": "<strong>",  
  "highlightPostTag": "</strong>",  
  "select": "HotelId, HotelName, Description, Category"  
}
```

Evaluate the response

Only the top 50 matches from the initial results can be semantically ranked. As with all queries, a response is composed of all fields marked as retrievable, or just those fields listed in the `select` parameter. A response includes the original relevance score, and might also include a count, or batched results, depending on how you formulated the request.

In semantic ranking, the response has more elements: a new [semantically ranked relevance score](#), an optional caption in plain text and with highlights, and an optional [answer](#). If your results don't include these extra elements, then your query might be misconfigured. As a first step towards troubleshooting the problem, check the semantic configuration to ensure it's specified in both the index definition and query.

In a client app, you can structure the search page to include a caption as the description of the match, rather than the entire contents of a specific field. This approach is useful when individual fields are too dense for the search results page.

The response for the above example query ("*interesting hotel with restaurant on site and cozy lobby or shared area*") returns three answers (`"answers": "extractive|count-e"`). Captions are returned because the "captions" property is set, with plain text and highlighted versions. If an answer can't be determined, it's omitted from the response. For brevity, this example shows just the three answers and the three highest scoring results from the query.

JSON

```
{  
  "@odata.count": 29,  
  "@search.answers": [  
    {  
      "HotelId": 1,  
      "HotelName": "The Grand Hotel",  
      "Description": "A large, historic hotel located in the heart of the city.",  
      "Category": "Luxury",  
      "Score": 0.95  
    },  
    {  
      "HotelId": 2,  
      "HotelName": "The Plaza Hotel",  
      "Description": "A modern hotel with a focus on comfort and convenience.",  
      "Category": "Mid-range",  
      "Score": 0.92  
    },  
    {  
      "HotelId": 3,  
      "HotelName": "The Royal Hotel",  
      "Description": "A classic hotel with a traditional atmosphere.",  
      "Category": "Budget",  
      "Score": 0.88  
    }  
  ]  
}
```

```
{  
    "key": "24",  
    "text": "Chic hotel near the city. High-rise hotel in downtown, within walking  
distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum  
by day, and then head over to Benaroya Hall to catch the evening's concert  
performance.",  
    "highlights": "Chic hotel near the city. High-rise hotel in downtown,  
</strong>within walking distance to </strong>theaters, art galleries, restaurants and shops.</strong> Visit Seattle Art Museum by day, and then  
head over to Benaroya Hall to catch the evening's concert performance.",  
    "score": 0.9340000152587891  
,  
{  
    "key": "40",  
    "text": "Only 8 miles from Downtown. On-site bar/restaurant, Free hot  
breakfast buffet, Free wireless internet, All non-smoking hotel. Only 15 miles from  
airport.",  
    "highlights": "Only 8 miles from Downtown. On-site bar/restaurant,  
Free hot breakfast buffet, Free wireless internet, </strong>All non-smoking hotel.</strong> Only 15 miles from airport.",  
    "score": 0.9210000038146973  
,  
{  
    "key": "38",  
    "text": "Nature is Home on the beach. Explore the shore by day, and then come  
home to our shared living space to relax around a stone fireplace, sip something  
warm, and explore the library by night. Save up to 30 percent. Valid Now through the  
end of the year. Restrictions and blackouts may apply.",  
    "highlights": "Nature is Home on the beach. Explore the shore by day, and then  
come home to our shared living space </strong>to relax around a stone  
fireplace, sip something warm, and explore the library by night. Save up to 30  
percent. Valid Now through the end of the year. Restrictions and blackouts may  
apply.",  
    "score": 0.9200000166893005  
}  
],  
"value": [  
{  
    "@search.score": 3.2328331,  
    "@search.rerankerScore": 2.575303316116333,  
    "@search.captions": [  
        {  
            "text": "The best of old town hospitality combined with views of the river  
and cool breezes off the prairie. Our penthouse suites offer views for miles and the  
rooftop plaza is open to all guests from sunset to 10 p.m. Enjoy a complimentary  
continental breakfast in the lobby, and free Wi-Fi throughout the hotel.",  
            "highlights": "The best of old town hospitality combined with views of the  
river and cool breezes off the prairie. Our penthouse </strong>suites offer  
views for miles and the rooftop plaza </strong>is open to all guests from  
sunset to 10 p.m. Enjoy a complimentary continental breakfast in the lobby,  
</strong>and free Wi-Fi throughout </strong>the hotel."  
        }  
    ],  
    "HotelId": "50",  
    "HotelName": "Head Wind Resort",
```

```
        "Description": "The best of old town hospitality combined with views of the river and cool breezes off the prairie. Our penthouse suites offer views for miles and the rooftop plaza is open to all guests from sunset to 10 p.m. Enjoy a complimentary continental breakfast in the lobby, and free Wi-Fi throughout the hotel.",  
        "Category": "Suite"  
    },  
    {  
        "@search.score": 0.632956,  
        "@search.rerankerScore": 2.5425150394439697,  
        "@search.captions": [  
            {  
                "text": "Every stay starts with a warm cookie. Amenities like the Counting Sheep sleep experience, our Wake-up glorious breakfast buffet and spacious workout facilities await.",  
                "highlights": "Every stay starts with a warm cookie. Amenities like the<strong> Counting Sheep sleep experience, </strong>our<strong> Wake-up glorious breakfast buffet and spacious workout facilities </strong>await."  
            }  
        ],  
        "HotelId": "34",  
        "HotelName": "Lakefront Captain Inn",  
        "Description": "Every stay starts with a warm cookie. Amenities like the Counting Sheep sleep experience, our Wake-up glorious breakfast buffet and spacious workout facilities await.",  
        "Category": "Budget"  
    },  
    {  
        "@search.score": 3.7076726,  
        "@search.rerankerScore": 2.4554927349090576,  
        "@search.captions": [  
            {  
                "text": "Chic hotel near the city. High-rise hotel in downtown, within walking distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum by day, and then head over to Benaroya Hall to catch the evening's concert performance.",  
                "highlights": "Chic hotel near the city. <strong>High-rise hotel in downtown, </strong>within<strong> walking distance to </strong>theaters, art<strong> galleries, restaurants and shops.</strong> Visit Seattle Art Museum by day, and then head over to Benaroya Hall to catch the evening's concert performance."  
            }  
        ],  
        "HotelId": "24",  
        "HotelName": "Uptown Chic Hotel",  
        "Description": "Chic hotel near the city. High-rise hotel in downtown, within walking distance to theaters, art galleries, restaurants and shops. Visit Seattle Art Museum by day, and then head over to Benaroya Hall to catch the evening's concert performance.",  
        "Category": "Suite"  
    },  
    . . .  
}
```

Expected workloads

For semantic ranking, you should expect a search service to support up to 10 concurrent queries per replica.

The service throttles semantic ranking requests if volumes are too high. An error message that includes these phrases indicate the service is at capacity for semantic ranking:

JSON

```
Error in search query: Operation returned an invalid status 'Partial Content'  
@search.semanticPartialResponseReason`  
CapacityOverloaded
```

If you anticipate consistent throughput requirements near, at, or higher than this level, please file a support ticket so that we can provision for your workload.

Next steps

Semantic ranking can be used in hybrid queries that combine keyword search and vector search into a single request and a unified response.

[Hybrid query with semantic ranker](#)

Last updated on 11/06/2025

Rewrite queries with semantic ranker in Azure AI Search (Preview)

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Query rewriting is the process of transforming a user's query into a more effective one, adding more terms and refining search results. The search service sends the search query (or a variation of it) to a generative model that generates alternative queries.

Query rewriting improves results from [semantic ranking](#) by correcting typos or spelling errors in user queries, and expanding queries with synonyms.

Search with query rewriting works like this:

- The user query is sent via the `search` property in the request.
- The search service sends the search query (or a variation of it) to a generative model that generates alternative queries.
- The search service uses the original query and the rewritten queries to retrieve search results.

Query rewriting is an optional feature. Without query rewriting, the search service just uses the original query to retrieve search results.

ⓘ Note

The rewritten queries might not contain all of the exact terms the original query had. This might impact search results if the query was highly specific and required exact matches for unique identifiers or product codes.

Prerequisites

- [Azure AI Search](#) in any [region that provides query rewrite](#), with [semantic ranker enabled](#).
- An existing search index with a [semantic configuration](#) and rich text content. The examples in this guide use the [hotels-sample-index](#) sample data to demonstrate query

rewriting.

- To follow the instructions in this article, you need a web client that supports REST API requests. The examples in this article were tested with [Visual Studio Code](#) and the [REST Client](#) extension.

 Tip

Content that includes explanations or definitions work best for semantic ranking.

Make a search request with query rewrites

In this REST API example, use [Search Documents \(preview\)](#) to formulate the request.

1. Paste the following request into a web client as a template.

HTTP

```
POST https://[search-service-name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-11-01-preview
{
    "search": "newer hotel near the water with a great restaurant",
    "semanticConfiguration": "en-semantic-config",
    "queryType": "semantic",
    "queryRewrites": "generative|count-5",
    "queryLanguage": "en-US",
    "debug": "queryRewrites",
    "top": 1
}
```

- Replace `search-service-name` with your search service name.
- Replace `hotels-sample-index` with your index name if it's different.
- Set "search" to a full text search query. The search property is required for query rewriting, unless you specify [vector queries](#). If you specify vector queries, then the "search" text must match the `"text"` property of the `"vectorQueries"` object. Your search string can support either the [simple syntax](#) or [full Lucene syntax](#).
- Set "semanticConfiguration" to a [predefined semantic configuration](#) embedded in your index.
- Set "queryType" to "semantic". You either need to set "queryType" to "semantic" or include a nonempty "semanticQuery" property in the request. [Semantic ranking](#) is required for query rewriting.

- Set "queryRewrites" to "generative|count-5" to get up to five query rewrites. You can set the count to any value between 1 and 10.
- Since you requested query rewrites by setting the "queryRewrites" property, you must set "queryLanguage" to the search text language. The search service uses the same language for the query rewrites. In this example, you use "en-US". The supported locales are: en-AU, en-CA, en-GB, en-IN, en-US, ar-EG, ar-JO, ar-KW, ar-MA, ar-SA, bg-BG, bn-IN, ca-ES, cs-CZ, da-DK, de-DE, el-GR, es-ES, es-MX, et-EE, eu-ES, fa-AE, fi-FI, fr-CA, fr-FR, ga-IE, gl-ES, gu-IN, he-IL, hi-IN, hr-BA, hr-HR, hu-HU, hy-AM, id-ID, is-IS, it-IT, ja-JP, kn-IN, ko-KR, lt-LT, lv-LV, ml-IN, mr-IN, ms-BN, ms-MY, nb-NO, nl-BE, nl-NL, no-NO, pa-IN, pl-PL, pt-BR, pt-PT, ro-RO, ru-RU, sk-SK, sl-SL, sr-BA, sr-ME, sr-RS, sv-SE, ta-IN, te-IN, th-TH, tr-TR, uk-UA, ur-PK, vi-VN, zh-CN, zh-TW.
- Set "debug" to "queryRewrites" to get the query rewrites in the response.

 **Tip**

Only set "debug": "queryRewrites" for testing purposes. For better performance, don't use debug in production.

- Set "top" to 1 to return only the top search result.

2. Send the request to execute the query and return results.

Next, you evaluate the search results with the query rewrites.

Evaluate the response

Here's an example of a response that includes query rewrites:

JSON

```
"@search.debug": {
  "semantic": null,
  "queryRewrites": {
    "text": {
      "inputQuery": "newer hotel near the water with a great restaurant",
      "rewrites": [
        "new waterfront hotels with top-rated eateries",
        "new waterfront hotels with top-rated restaurants",
        "new waterfront hotels with excellent dining",
        "new waterfront hotels with top-rated dining",
        "new water-side hotels with top-rated restaurants"
      ]
    }
  }
}
```

```

        ],
      },
      "vectors": []
    },
  },
  "value": [
    {
      "@search.score": 58.992092,
      "@search.rerankerScore": 2.815633535385132,
      "HotelId": "18",
      "HotelName": "Ocean Water Resort & Spa",
      "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near the pier, rooftop pool, waterfront dining & more.",
      "Description_fr": "Nouvel h\u00f4tel de luxe pour des vacances inoubliables. Vue sur la baie depuis chaque chambre, emplacement pr\u00e8s de la jet\u00e9e, piscine sur le toit, restaurant au bord de l'eau et plus encore.",
      "Category": "Luxury",
      "Tags": [
        "view",
        "pool",
        "restaurant"
      ],
      "ParkingIncluded": true,
      "LastRenovationDate": "2020-11-14T00:00:00Z",
      "Rating": 4.2,
      "Location": {
        "type": "Point",
        "coordinates": [
          -82.537735,
          27.943701
        ],
        "crs": {
          "type": "name",
          "properties": {
            "name": "EPSG:4326"
          }
        }
      },
      //... more properties redacted for brevity
    }
  ]
]

```

Here are some key points to note:

- Because you set the "debug" property to "queryRewrites" for testing, the response includes a `@search.debug` object with the text input query and query rewrites.
- Because you set the "queryRewrites" property to "generative|count-5", the response includes up to five query rewrites.
- The `"inputQuery"` value is the query sent to the generative model for query rewriting. The input query isn't always the same as the user's `"search"` query.

Here's an example of a response without query rewrites.

JSON

```
"@search.debug": {
  "semantic": null,
  "queryRewrites": {
    "text": {
      "inputQuery": "",
      "rewrites": []
    },
    "vectors": []
  }
},
"value": [
  {
    "@search.score": 7.774868,
    "@search.rerankerScore": 2.815633535385132,
    "HotelId": "18",
    "HotelName": "Ocean Water Resort & Spa",
    "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views from every room, location near the pier, rooftop pool, waterfront dining & more.",
    "Description_fr": "Nouvel h\u00f4tel de luxe pour des vacances inoubliables. Vue sur la baie depuis chaque chambre, emplacement pr\u00e8s de la jet\u00e9e, piscine sur le toit, restaurant au bord de l'eau et plus encore.",
    "Category": "Luxury",
    "Tags": [
      "view",
      "pool",
      "restaurant"
    ],
    "ParkingIncluded": true,
    "LastRenovationDate": "2020-11-14T00:00:00Z",
    "Rating": 4.2,
    "Location": {
      "type": "Point",
      "coordinates": [
        -82.537735,
        27.943701
      ],
      "crs": {
        "type": "name",
        "properties": {
          "name": "EPSG:4326"
        }
      }
    },
    //... more properties redacted for brevity
  }
]
```

Vector queries with query rewrite

You can include vector queries in your search request to combine keyword search and vector search into a single request and a unified response.

Here's an example of a query that includes a vector query with query rewrites. Modify a [previous example](#) to include a vector query.

- Add a "vectorQueries" object to the request. This object includes a vector query with the "kind" set to "text".
- The "text" value is the same as the "search" value. These values must be identical for query rewriting to work.

HTTP

```
POST https://[search-service-name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2025-11-01-preview
{
    "search": "newer hotel near the water with a great restaurant",
    "vectorQueries": [
        {
            "kind": "text",
            "text": "newer hotel near the water with a great restaurant",
            "k": 50,
            "fields": "Description",
            "queryRewrites": "generative|count-3"
        }
    ],
    "semanticConfiguration": "en-semantic-config",
    "queryType": "semantic",
    "queryRewrites": "generative|count-5",
    "queryLanguage": "en-US",
    "top": 1
}
```

The response includes query rewrites for both the text query and the vector query.

Test query rewrites with debug

You should test your query rewrites to ensure that they're working as expected. Set the `"debug": "queryRewrites"` property in your query request to get the query rewrites in the response. Setting `"debug"` is optional for testing purposes. For better performance, don't set this property in production.

Partial response reasons

You might observe that the debug (test) response includes an empty array for the `text.rewrites` and `vectors` properties.

JSON

```
{  
    "@odata.context": "https://demo-search-svc.search.windows.net/indexes('hotels-sample-index')/$metadata#docs(*)",  
    "@search.debug": {  
        "semantic": null,  
        "queryRewrites": {  
            "text": {  
                "rewrites": []  
            },  
            "vectors": []  
        }  
    },  
    "@search.semanticPartialResponseReason": "Transient",  
    "@search.semanticQueryRewriteResultType": "OriginalQueryOnly",  
    //... more properties redacted for brevity  
}
```

In the preceding example:

- The response includes a `@search.semanticPartialResponseReason` property with a value of "Transient". This message means that at least one of the queries failed to complete.
- The response also includes a `@search.semanticQueryRewriteResultType` property with a value of "OriginalQueryOnly". This message means that the query rewrites are unavailable. Only the original query is used to retrieve search results.

Next steps

Semantic ranking can be used in hybrid queries that combine keyword search and vector search into a single request and a unified response.

[Hybrid query with semantic ranker](#)

Last updated on 11/21/2025

Migrate semantic ranking code from previous versions

If your semantic ranking code was written against early preview APIs, this article identifies the code changes necessary for migrating to newer API versions. Breaking changes for semantic ranker are limited to query logic in recent APIs, but if your code was written against the initial preview version, you might need to change your semantic configuration as well.

Breaking changes

There are two breaking changes for semantic ranker across REST API versions:

- `searchFields` was replaced by `semanticConfiguration` in 2021-04-30-preview
- `queryLanguage` was ignored starting in 2023-07-01-preview, but reinstated for query rewrite in 2024-11-01-preview

Other version-specific updates pertain to new capabilities, but don't break existing code and are therefore not breaking changes.

If you're using Azure SDKs, multiple APIs have been renamed over time. The SDK change logs provide the details.

API versions providing semantic ranking

Check your code for the REST API version or SDK package version to confirm which one provides semantic ranking. The following API versions have some level of support for semantic ranking.

 Expand table

Release type	REST API version	Semantic ranker updates
initial	2020-06-30-preview	Adds <code>queryType=semantic</code> to Search Documents
preview	2021-04-30-preview	Adds <code>semanticConfiguration</code> to Create or Update Index
preview	2023-07-01-preview	Updates <code>semanticConfiguration</code> . Starting on July 14, 2023 updates to the Microsoft-hosted semantic models made semantic ranker language-agnostic, effectively decommissioning the <code>queryLanguage</code> property for semantic ranking. There's no breaking change in code,

Release type	REST API version	Semantic ranker updates
		but the property is ignored. Customers were advised to remove this property from code.
preview	2023-10-01-preview	Adds <code>semanticQuery</code> to send a query used only for reranking purposes.
stable	2023-11-01	Generally available. Introduced changes to <code>semanticConfiguration</code> that progressed to the stable version. If your code targets this version or later, it's compatible with newer API versions unless you adopt new preview features.
preview	2024-05-01-preview	No change
stable	2024-07-01	No change
preview	2024-09-01-preview	No change
preview	2024-11-01-preview	Adds query rewrite. The <code>queryLanguage</code> property is now required if you use query rewrite (preview) .
preview	2025-03-01-preview	Adds opt-in to prerelease versions of semantic models.
preview	2025-05-01-preview	No API updates in this preview, but semantic ranking now has better integration with scoring profiles .
preview	2025-08-01-preview	No change
preview	2025-11-01-preview	Available on free tiers.

Change logs for Azure SDKs

Azure SDKs are on an independent release schedule. You should check the change logs to determine which packages provide semantic features and whether any APIs have been renamed.

- [Azure SDK for .NET change log ↗](#)
- [Azure SDK for Python change log ↗](#)
- [Azure SDK for Java change log ↗](#)
- [Azure SDK for JavaScript change log ↗](#)

2024-11-01-preview

- Adds [query rewrite](#) to Search Documents.
- Requires `queryLanguage` for query rewrite workloads. For a list of valid values, see the [REST API](#).

2024-09-01-preview

No changes to semantic ranking syntax from the 2024-07-01 stable version.

2024-07-01

No changes to semantic ranking syntax from the 2024-05-01-preview version.

Don't use this API version. It implements a vector query syntax that's incompatible with any newer API version.

2024-05-01-preview

No changes to semantic ranking syntax from the 2024-03-01-preview version.

2024-03-01-preview

No changes to semantic ranking syntax from the 2023-10-01-preview version, but vector queries are introduced. Semantic ranking now applies to responses from hybrid and vector queries. You can apply reranking on any human-readable text fields in the response, assuming the fields are listed in `prioritizedFields`.

2023-11-01

- Excludes `SemanticDebug` and `semanticQuery`, otherwise the same as the 2023-10-01-preview version.

2023-10-01-preview

- Adds `semanticQuery`

2023-07-01-preview

- Adds `semanticErrorHandling`, `semanticMaxWaitInMilliseconds`.
- Adds numerous semantic-related fields to the response, such as `SemanticDebug` and `SemanticErrorMode`.
- Ignores `queryLanguage`, it's no longer used in semantic ranking.

Starting on July 14, 2023, semantic ranker is language agnostic. In preview versions, semantic ranking would deprioritize results differing from the `querylanguage` specified by the field analyzer. However, the `queryLanguage` property is still applicable to [spell correction](#) and the short list of languages supported by that feature.

2021-04-30-preview

- Semantic support is through [Search Documents](#) and [Create or Update Index](#) preview API calls.
- Adds `semanticConfiguration` to a search index. A semantic configuration has a name and a prioritized field list.
- Adds ``prioritizedFields`.

The `searchFields` property is no longer used to prioritize fields. In all versions moving forward, `semanticConfiguration.prioritizedFields` replaces `searchFields` as the mechanism for specifying which fields to use for L2 ranking.

2020-06-30-preview

- Semantic support is through a [Search Documents](#) preview API call.
- Adds `queryType=semantic` to the query request.
- Adapts `searchFields` so that if the query type is semantic, the `searchFields` property determines the priority order of field inputs to the semantic ranker.
- Adds `captions`, `answers`, and `highlights` to the query response.

Next steps

Test your semantic configuration migration by running a semantic query.

[Create a semantic query](#)

Use scoring profiles with semantic ranker in Azure AI Search

09/28/2025

You can apply a [scoring profile](#) over [semantically ranked search results](#), where the scoring profile is processed last.

To ensure the scoring profile provides the determining score, the semantic ranker adds a response field, `@search.rerankerBoostedScore`, that applies scoring profile logic on semantically ranked results. In search results that include `@search.score` from level 1 ranking, `@search.rerankerScore` from semantic ranker, and `@search.reRankerBoostedScore`, results are sorted by `@search.reRankerBoostedScore`.

Prerequisites

- [Azure AI Search](#), Basic pricing tier or higher, with [semantic ranker](#) enabled.
- A search index with a semantic configuration that specifies `"rankingOrder": "boostedRerankerScore"` and a scoring profile that specifies [functions](#).

Limitations

Boosting of semantically ranked results applies to scoring profile functions only. There's no boosting if the scoring profile consists only of weighted text fields.

How does semantic configuration with scoring profiles work?

When you execute a semantic query associated with a scoring profile, a third search score, `@search.rerankerBoostedScore` value, is generated for every document in your search results. This boosted score, calculated by applying the scoring profile to the existing reranker score, doesn't have a guaranteed range (0–4) like a normal reranker score, and scores can be significantly higher than 4.

Semantic results are sorted by `@search.rerankerBoostedScore` by default if a scoring profile exists. If the `rankingOrder` property isn't specified, then `BoostedRerankerScore` is the default value in the semantic configuration.

In this scenario, a scoring profile is used twice.

1. First, the scoring profile defined in your index is used during the initial L1 ranking phase, boosting results from:
 - Text-based queries (BM25 or RRF)
 - The text portion of vector queries
 - Hybrid queries that combine both types
2. Next, the semantic ranker rescores the top 50 results, promoting more semantically relevant matches to the top. This step can erase the benefit of the scoring profile. For example, if you boosted based on freshness, then semantic reordering replaces that boost with its own logic of what is most relevant.
3. Finally, the scoring profile is applied again, after reranking, restoring the boosts influence over the final order of results. If you boost by freshness, the semantically ranked results are rescored based on freshness.

Enable scoring profiles in semantic configuration

To enable scoring profiles for semantically ranked results, [update an index](#) by setting the `rankingOrder` property of its semantic configuration. Use the PUT method to update the index with your revisions. No index rebuild is required.

JSON

```
PUT https://{{service-name}}.search.windows.com/indexes/{{index-name}}?api-version=2025-09-01
{
  "semantic": {
    "configurations": [
      {
        "name": "mySemanticConfig",
        "rankingOrder": "boostedRerankerScore"
      }
    ]
  }
}
```

Disable scoring profiles in semantic configuration

To opt out of sorting by semantic reranker boosted score, set the `rankingOrder` field to `reRankerScore` value in the semantic configuration.

JSON

```
PUT /indexes/{index-name}?api-version=2025-09-01
{
  "semantic": {
    "configurations": [
      {
        "name": "mySemanticConfig",
        "rankingOrder": "reRankerScore"
      }
    ]
  }
}
```

Even if you opt out of sorting by `@search.rerankerBoostedScore`, the `boostedRerankerScore` field is still produced in the response, but it's no longer used to sort results.

Example query and response

Start with a [semantic query](#) that specifies a scoring profile. This query targets a search index that has `rankingOrder` set to `boostedRerankerScore`.

JSON

```
POST /indexes/{index-name}/docs/search?api-version=2025-09-01
{
  "search": "my query to be boosted",
  "scoringProfile": "myScoringProfile",
  "queryType": "semantic"
}
```

The response includes the new `rerankerBoostedScore`, alongside the L1 `@search.score` and the L2 `@search.rerankerScore`. Results are ordered by `@search.rerankerBoostedScore`.

JSON

```
{
  "value": [
    {
      "@search.score": 0.63,
      "@search.rerankerScore": 2.98,
      "@search.rerankerBoostedScore": 7.68,
      "content": "boosted content 2"
    },
    {
      "@search.score": 1.12,
      "@search.rerankerScore": 3.12,
      "@search.rerankerBoostedScore": 5.61,
    }
  ]
}
```

```
"content": "boosted content 1"
```

```
}
```

```
]
```

```
}
```

Configure network access and firewall rules for Azure AI Search

This article explains how to restrict network access to a search service's public endpoint. To block *all* data plane access to the public endpoint, use [private endpoints](#) and connect from within an Azure virtual network.

This article assumes the Azure portal for configuring network access options. You can also use the [Management REST API](#), [Azure PowerShell](#), or the [Azure CLI](#).

Prerequisites

- A search service, any region, at the Basic tier or higher
- Owner or Contributor permissions

Limitations

There are drawbacks to locking down the public endpoint:

- It takes time to fully identify IP ranges and set up firewalls, and if you're in early stages of proof-of-concept testing and investigation and using sample data, you might want to defer network access controls until you actually need them.
- Some workflows require access to a public endpoint. Specifically, the [indexing wizards](#) in the Azure portal connect to built-in (hosted) sample data and embedding models over the public endpoint. You can switch to code or script to complete the same tasks when firewall rules in place, but if you want to run the wizards, the public endpoint must be available. For more information, see [Secure connections in the import wizards](#).

When to configure network access

By default, Azure AI Search is configured to allow connections over a public endpoint. Access to a search service *through* the public endpoint is protected by authentication and authorization protocols, but the endpoint itself is open to the internet at the network layer for data plane requests.

If you aren't hosting a public web site, you might want to configure network access to automatically refuse requests unless they originate from an approved set of devices and cloud services.

There are two mechanisms for restricting access to the public endpoint:

- Inbound rules listing the IP addresses, ranges, or subnets from which requests are admitted
- Exceptions to network rules, where requests are admitted with no checks, as long as the request originates from a [trusted service](#)

Network rules aren't required, but it's a security best practice to add them if you use Azure AI Search for surfacing private or internal corporate content.

Network rules are scoped to data plane operations against the search service's public endpoint. Data plane operations include creating or querying indexes, and all other actions described by the [Search REST APIs](#). Control plane operations target service administration. Those operations specify resource provider endpoints, which are subject to the [network protections supported by Azure Resource Manager](#).

Configure network access in Azure portal

1. Sign in to Azure portal and [find your search service](#).
2. Under **Settings**, select **Networking** on the leftmost pane. If you don't see this option, check your service tier. Networking options are available on the Basic tier and higher.
3. Choose **Selected IP addresses**. Avoid the **Disabled** option unless you're configuring a [private endpoint](#).

The screenshot shows the 'Networking' settings page in the Azure portal. The 'IP Firewall' section is selected. At the top, there are three tabs: 'Firewalls and virtual networks' (which is selected), 'Private endpoint connections', and 'Shared private access'. Below the tabs, a note states: 'Public endpoints allow access to this resource through the internet using a public IP address. An application or resource that is granted access with the following network rules still requires proper authorization to access this resource.' Under 'Public network access', there are three options: 'All networks' (radio button is empty), 'Selected IP addresses' (radio button is checked and highlighted with a red box), and 'Disabled' (radio button is empty). To the right of the radio buttons is a magnifying glass icon with a plus sign inside, used for adding new IP addresses.

4. Under **IP Firewall**, select **Add your client IP address**. This step creates an inbound rule for the public IP address of your personal device to Azure AI Search. See [Allow access from the Azure portal IP address](#) for details.

IP firewall

Add IP ranges to restrict access to specific Internet-based services or on-premises networks and block general Internet traffic. [Learn more](#)

 Add your client IP address

Restricted address range

IP address or CIDR



5. Add other client IP addresses for other devices and services that send requests to a search service.

Specify IP addresses and ranges in the CIDR format. An example of CIDR notation is 8.8.8.0/24, which represents the IPs that range from 8.8.8.0 to 8.8.8.255.

To get the public IP addresses of Azure services, see [Azure IP Ranges and Service Tags](#). If your search client is hosted within an Azure function, see [IP addresses in Azure Functions](#).

6. Under **Exceptions**, select **Allow Azure services on the trusted services list to access this search service**.

Exceptions

Allow Azure services on the trusted services list to access this search service.

The trusted service list includes:

- `Microsoft.CognitiveServices` for Azure OpenAI and Foundry Tools
- `Microsoft.MachineLearningServices` for Azure Machine Learning

When you enable this exception, you take a dependency on Microsoft Entra ID authentication, managed identities, and role assignments. Any Foundry Tool or AML feature that has a valid role assignment on your search service can bypass the firewall. See [Grant access to trusted services](#) for more details.

7. Save your changes.

After you enable the IP access control policy for your Azure AI Search service, all requests to the data plane from machines outside the allowed list of IP address ranges are rejected.

When requests originate from IP addresses that aren't in the allowed list, a generic **403 Forbidden** response is returned with no other details.

Important

It can take several minutes for changes to take effect. Wait at least 15 minutes before troubleshooting any problems related to network configuration.

Allow access from the Azure portal IP address

The Azure portal has its own connection to Azure AI Search, separate from your local device and browser. If you use the Azure portal to manage your search service, you need to add the portal IP address as described in this section, and your client IP address as described in the previous section.

When IP rules are configured, some features of the Azure portal are disabled. For example, you can view and manage service level information, but portal access to the import wizards, indexes, indexers, and other top-level resources are restricted.

You can restore the Azure portal's access to the full range of search service operations by adding the Azure portal IP address to the restricted address range.

To get the Azure portal's IP address, perform `nslookup` (or `ping`) on:

- `stamp2.ext.search.windows.net`, which is the domain of the traffic manager for the Azure public cloud.
- `stamp2.ext.search.azure.us` for Azure Government cloud.

For `nslookup`, the IP address is visible in the "Non-authoritative answer" portion of the response. In the following example, the IP address that you should copy is `52.252.175.48`.

Bash

```
$ nslookup stamp2.ext.search.windows.net
Server: ZenWiFi_ET8-0410
Address: 192.168.50.1

Non-authoritative answer:
Name: azsyrie.northcentralus.cloudapp.azure.com
Address: 52.252.175.48
Aliases: stamp2.ext.search.windows.net
          azs-ux-prod.trafficmanager.net
          azspncuux.management.search.windows.net
```

When services run in different regions, they connect to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

For ping, the request times out, but the IP address is visible in the response. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is 52.252.175.48.

A banner informs you that IP rules affect the Azure portal experience. This banner remains visible even after you add the Azure portal's IP address. Remember to wait several minutes for network rules to take effect before testing.



ⓘ **Restricted access:** When IP rules are configured, some features of the Azure portal may be disabled. You can view and manage service level information, but portal access to indexes, indexers, and other top-level resources is restricted. You can restore portal access to the full range of search service operations by allowing access from your client IP address. [Learn how to continue using your service fully through the portal](#)

Grant access to trusted Azure services

Did you select the trusted services exception? If yes, your search service admits requests and responses from a trusted Azure resource without checking for an IP address. A trusted resource must have a managed identity (either system or user-assigned, but usually system). A trusted resource must have a role assignment on Azure AI Search that gives it permission to data and operations.

The trusted service list for Azure AI Search includes:

- Microsoft.CognitiveServices for Azure OpenAI and Foundry Tools
- Microsoft.MachineLearningServices for Azure Machine Learning

Workflows for this network exception are requests originating from Microsoft Foundry or other AML features to Azure AI Search. The trusted services exception is typically for [Azure OpenAI On Your Data](#) scenarios for retrieval augmented generation (RAG) and playground environments.

Trusted resources must have a managed identity

To set up managed identities for Azure OpenAI and Azure Machine Learning:

- [How to configure Azure OpenAI in Foundry Models with managed identities](#)
- [How to set up authentication between Azure Machine Learning and other services.](#)

To set up a managed identity for a Foundry resource:

1. [Find your Foundry resource](#).
2. From the left pane, select **Resource management > Identity**.
3. Set **System assigned** to **On**.

Trusted resources must have a role assignment

Once your Azure resource has a managed identity, [assign roles on Azure AI Search](#) to grant permissions to data and operations.

The trusted services are used for vectorization workloads: generating vectors from text and image content, and sending payloads back to the search service for query execution or indexing. Connections from a trusted service are used to deliver payloads to Azure AI search.

1. [Find your search service](#).
2. On the leftmost pane, under **Access control (IAM)**, select **Identity**.
3. Select **Add** and then select **Add role assignment**.
4. On the **Roles** page:
 - Select **Search Index Data Contributor** to load a search index with vectors generated by an embedding model. Choose this role if you intend to use integrated vectorization during indexing.
 - Or, select **Search Index Data Reader** to provide queries containing a vector generated by an embedding model at query time. The embedding used in a query isn't written to an index, so no write permissions are required.
5. Select **Next**.
6. On the **Members** page, select **Managed identity** and **Select members**.
7. Filter by system-managed identity and then select the managed identity of your Foundry resource.

(!) Note

This article covers the trusted exception for admitting requests to your search service, but Azure AI Search is itself on the trusted services list of other Azure resources. Specifically, you can use the trusted service exception for [connections from Azure AI Search to Azure Storage](#).

Next steps

Once a request is allowed through the firewall, it must be authenticated and authorized. You have two options:

- [Key-based authentication](#), where an admin or query API key is provided on the request. This option is the default.
- [Role-based access control](#) using Microsoft Entra ID, where the caller is a member of a security role on a search service. This is the most secure option. It uses Microsoft Entra ID for authentication and role assignments on Azure AI Search for permissions to data and operations.

[Enable RBAC on your search service](#)

Last updated on 11/18/2025

Create a private endpoint for a secure connection to Azure AI Search

09/23/2025

This article explains how to configure a private connection to Azure AI Search so that it admits requests from clients in a virtual network instead of over a public internet connection:

- [Create an Azure virtual network](#), or use an existing one
- [Configure a search service to use a private endpoint](#)
- [Create an Azure virtual machine \(client\) in the same virtual network](#)
- [Test using a browser session on the virtual machine](#)

Other Azure resources that might privately connect to Azure AI Search include Azure OpenAI for "use your own data" scenarios. Azure AI Foundry doesn't run in a virtual network, but it can be configured on the backend to send requests over the Microsoft backbone network. Configuration for this traffic pattern is enabled by Microsoft when your request is submitted and approved. For this scenario:

- Follow the instructions in this article to set up the private endpoint.
- [Enable trusted service](#) of your search resource from the Azure portal.
- Optionally, [disable public network access](#) if connections should only originate from clients in virtual network or from Azure OpenAI over a private endpoint connection.

Key points about private endpoints

Private endpoints are provided by [Azure Private Link](#), as a separate billable service. For more information about costs, see [Azure Private Link pricing](#).

Once a search service has a private endpoint, portal access to that service must be initiated from a browser session on a virtual machine inside the virtual network. See [this step](#) for details.

You can create a private endpoint for a search service in the Azure portal, as described in this article. Alternatively, you can use the [Management REST API](#), [Azure PowerShell](#), or the [Azure CLI](#).

Why use a private endpoint?

[Private endpoints](#) for Azure AI Search allow a client on a virtual network to securely access data in a search index over a [Private Link](#). The private endpoint uses an IP address from the [virtual network address space](#) for your search service. Network traffic between the client and the

search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. For a list of other PaaS services that support Private Link, check the [availability section](#) in the product documentation.

Private endpoints for your search service allow you to:

- Block all connections on the public endpoint for your search service.
- Increase security for the virtual network, by letting you block exfiltration of data from the virtual network.
- Securely connect to your search service from on-premises networks that connect to the virtual network using [VPN](#) or [ExpressRoutes](#) with private-peering.

Create the virtual network

In this section, you create a virtual network and subnet to host the VM that will be used to access your search service's private endpoint.

1. From the Azure portal home tab, select **Create a resource > Networking > Virtual network**.
2. In **Create virtual network**, enter or select the following values:

 [Expand table](#)

Setting	Value
Subscription	Select your subscription
Resource group	Select Create new , enter a name, such as <i>myResourceGroup</i> , then select OK
Name	Enter a name, such as <i>MyVirtualNetwork</i>
Region	Select a region

3. Accept the defaults for the rest of the settings. Select **Review + create** and then **Create**.

Create a search service with a private endpoint

In this section, you create a new Azure AI Search service with a private endpoint.

1. On the upper-left side of the screen in the Azure portal, select **Create a resource > AI + machine learning > AI Search**.
2. In **Create a search service - Basics**, enter or select the following values:

[+] Expand table

Setting	Value
PROJECT DETAILS	
Subscription	Select your subscription
Resource group	Use the resource group that you created in the previous step
INSTANCE DETAILS	
URL	Enter a unique name
Location	Select your region
Pricing tier	Select Change Pricing Tier and choose your desired service tier. Private endpoints aren't supported on the Free tier. You must select Basic or higher.

3. Select **Next: Scale**.

4. Accept the defaults and select **Next: Networking**.

5. In **Create a search service - Networking**, select **Private** for **Endpoint connectivity (data)**.

6. Select **+ Add** under **Private endpoint**.

7. In **Create private endpoint**, enter or select values that associate your search service with the virtual network you created:

[+] Expand table

Setting	Value
Subscription	Select your subscription
Resource group	Use the resource group that you created in the previous step
Location	Select a region
Name	Enter a name, such as <i>myPrivateEndpoint</i>
Target subresource	Accept the default searchService
NETWORKING	
Virtual network	Select the virtual network you created in the previous step
Subnet	Select the default

Setting	Value
PRIVATE DNS INTEGRATION	
Enable Private DNS Integration	Select the checkbox
Private DNS zone	Accept the default (New) <code>privatelink.search.windows.net</code>

8. Select **Add**.
9. Select **Review + create**. You're taken to the **Review + create** page where Azure validates your configuration.
10. When you see the **Validation passed** message, select **Create**.
11. Once provisioning of your new service is complete, browse to the resource that you created.
12. Select **Settings > Keys** from the left content menu.
13. Copy the **Primary admin key** for later, when connecting to the service.

Create a virtual machine

1. On the upper-left side of the screen in the Azure portal, select **Create a resource > Compute > Virtual machine**.
2. In **Create a virtual machine - Basics**, enter or select the following values:

[] [Expand table](#)

Setting	Value
PROJECT DETAILS	
Subscription	Select your subscription
Resource group	Use the resource group that you created in the previous section
INSTANCE DETAILS	
Virtual machine name	Enter a name, such as <i>my-vm</i>
Region	Select your region
Availability options	You can choose No infrastructure redundancy required , or select another option if you need the functionality

Setting	Value
Image	Select Windows Server 2022 Datacenter: Azure Edition - Gen2
VM architecture	Accept the default x64
Size	Accept the default Standard D2S v3
ADMINISTRATOR ACCOUNT	
Username	Enter the user name of the administrator. Use an account that's valid for your Azure subscription. Sign in to the Azure portal from the VM so that you can manage your search service.
Password	Enter the account password. The password must be at least 12 characters long and meet the defined complexity requirements .
Confirm Password	Reenter password
INBOUND PORT RULES	
Public inbound ports	Accept the default Allow selected ports
Select inbound ports	Accept the default RDP (3389)

3. Select **Next: Disks**.

4. In **Create a virtual machine - Disks**, accept the defaults and select **Next: Networking**.

5. In **Create a virtual machine - Networking**, provide the following values:

[] [Expand table](#)

Setting	Value
Virtual network	Select the virtual network you created in a previous step
Subnet	Accept the default 10.1.0.0/24
Public IP	Accept the default
NIC network security group	Accept the default Basic
Public inbound ports	Select the default Allow selected ports
Select inbound ports	Select HTTP 80, HTTPS (443), and RDP (3389)

! **Note**

IPv4 addresses can be expressed in [CIDR](#) format. Remember to avoid the IP range reserved for private networking, as described in [RFC 1918](#):

- 10.0.0.0 - 10.255.255.255 (10/8 prefix)
- 172.16.0.0 - 172.31.255.255 (172.16/12 prefix)
- 192.168.0.0 - 192.168.255.255 (192.168/16 prefix)

6. Select **Review + create** for a validation check.

7. When you see the **Validation passed** message, select **Create**.

Connect to the VM

Download and then connect to the virtual machine as follows:

1. In the Azure portal's search bar, search for the virtual machine created in the previous step.
2. Select **Connect**. After selecting the **Connect** button, **Connect to virtual machine** opens.
3. Select **Download RDP File**. Azure creates a Remote Desktop Protocol (.rdp) file and downloads it to your computer.
4. Open the downloaded .rdp file.
 - a. If prompted, select **Connect**.
 - b. Enter the username and password you specified when creating the VM.

Note

You might need to select **More choices > Use a different account**, to specify the credentials you entered when you created the VM.

5. Select **OK**.

6. You might receive a certificate warning during the sign-in process. If you receive a certificate warning, select **Yes** or **Continue**.
7. Once the VM desktop appears, minimize it to go back to your local desktop.

Test connections

In this section, you verify private network access to the search service and connect privately to the using the Private Endpoint.

When the search service endpoint is private, some portal features are disabled. You can view and manage service level settings, but portal access to index data and various other components in the service, such as the index, indexer, and skillset definitions, is restricted for security reasons.

1. In the Remote Desktop of *myVM*, open PowerShell.
2. Enter `nslookup [search service name].search.windows.net`.

You'll receive a message similar to this:



```
Server: UnKnown
Address: 168.63.129.16
Non-authoritative answer:
Name: [search service name].privatelink.search.windows.net
Address: 10.0.0.5
Aliases: [search service name].search.windows.net
```

3. From the VM, connect to the search service and create an index. You can follow this [quickstart](#) to create a new search index in your service using the REST API. Setting up requests from a Web API test tool requires the search service endpoint (`https://[search service name].search.windows.net`) and the admin api-key you copied in a previous step.
4. Completing the quickstart from the VM is your confirmation that the service is fully operational.
5. Close the remote desktop connection to *myVM*.
6. To verify that your service isn't accessible on a public endpoint, open a REST client on your local workstation and attempt the first several tasks in the quickstart. If you receive an error that the remote server doesn't exist, you successfully configured a private endpoint for your search service.

Use the Azure portal to access a private search service

When the search service endpoint is private, some portal features are disabled. You can view and manage service level information, but index, indexer, and skillset information are hidden for security reasons.

To work around this restriction, connect to Azure portal from a browser on a virtual machine inside the virtual network. the Azure portal uses the private endpoint on the connection and gives you visibility into content and operations.

1. Follow the [steps to provision a VM that can access the search service through a private endpoint](#).
2. On a virtual machine in your virtual network, open a browser and sign in to the Azure portal. the Azure portal uses the private endpoint attached to the virtual machine to connect to your search service.

Disable public network access

You can lock down a search service to prevent it from admitting any request from the public internet. You can use the Azure portal for this step.

1. In the Azure portal, on the leftmost pane of your search service page, select **Networking**.
2. Select **Disabled** on the **Firewalls and virtual networks** tab.

You can also use the [Azure CLI](#), [Azure PowerShell](#), or the [Management REST API](#), by setting `public-access` or `public-network-access` to `disabled`.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money.

You can delete individual resources or the resource group to delete everything you created in this exercise. Select the resource group on any resource's overview page, and then select **Delete**.

Next step

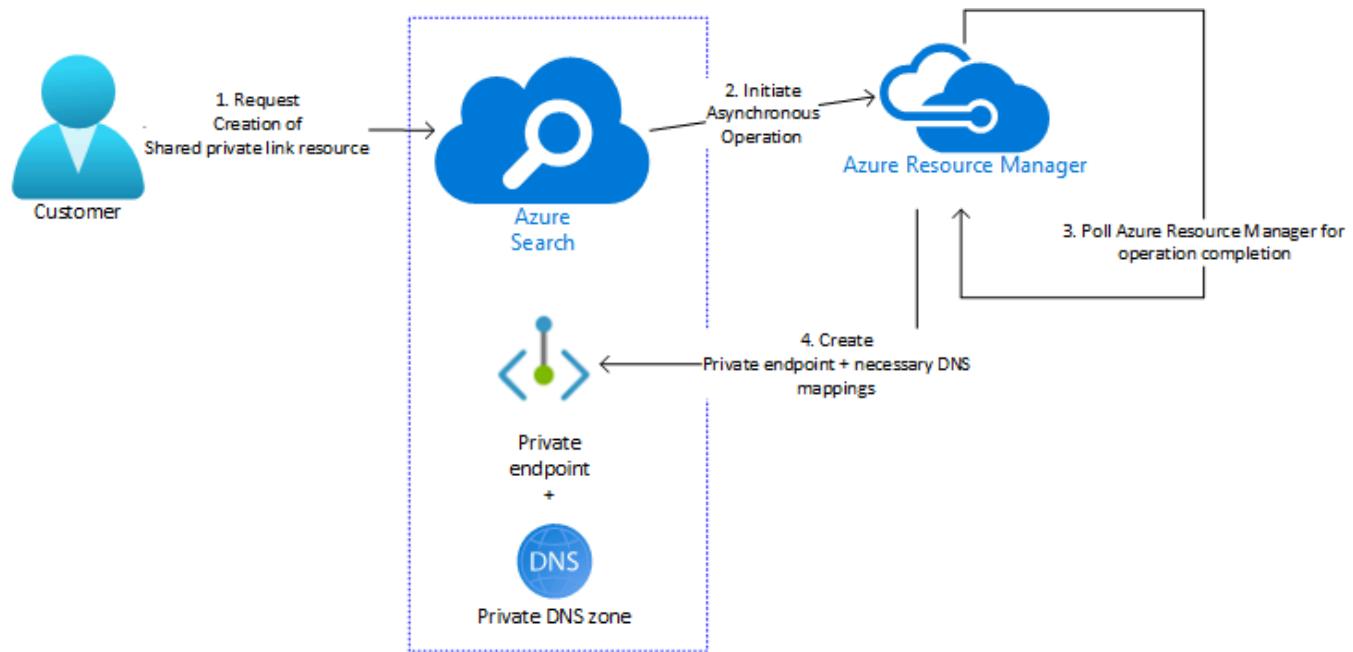
In this article, you created a VM on a virtual network and a search service with a private endpoint. You connected to the VM from the internet and securely communicated to the search service using Private Link. To learn more about private endpoints, see [What is a private endpoint?](#)

Troubleshoot issues with shared private links in Azure AI Search

06/04/2025

A shared private link allows Azure AI Search to make secure outbound connections over a private endpoint when accessing customer resources in a virtual network. This article can help you resolve errors that might occur.

Creating a shared private link is a search service control plane operation. You can [create a shared private link](#) using either the Azure portal or a [Management REST API](#). During provisioning, the state of the request is `Updating`. After the operation completes successfully, status is `Succeeded`. A private endpoint to the resource, along with any DNS zones and mappings, is created. This endpoint is used exclusively by your search service instance and is managed through Azure AI Search.



The following are common errors that occur during the creation phase.

Request validation failures

- **Unsupported SKU:** Shared private links are supported on the Basic tier and higher. For indexers with skillsets, the minimum tier is Standard 1 (S1). For more information, see [Shared private link resource limits](#).
- **Invalid name:** Naming rules for a shared private link are:
 - Length should be between 1 to 60 characters
 - Alphanumeric characters

- Names can include underscore `_`, period `.`, and hyphen `-` as long as it's not the first character in the name
- Invalid group ID: Group IDs are case-sensitive and must be one of the following values.

[\[+\] Expand table](#)

Azure resource	Group ID	First available API version
Azure Storage - Blob (or) ADLS Gen 2	<code>blob</code>	<code>2020-08-01</code>
Azure Storage - Tables	<code>table</code>	<code>2020-08-01</code>
Azure Cosmos DB for NoSQL	<code>Sql</code>	<code>2020-08-01</code>
Azure SQL Database	<code>sqlServer</code>	<code>2020-08-01</code>
Azure Database for MySQL (preview)	<code>mysqlServer</code>	<code>2020-08-01-Preview</code>
Azure Key Vault	<code>vault</code>	<code>2020-08-01</code>
Azure Functions (preview)	<code>sites</code>	<code>2020-08-01-Preview</code>

Resources marked with "(preview)" must be created using a preview version of the Management REST API versions.

- `privateLinkResourceId` type validation: Similar to `groupId`, Azure AI Search validates that the "correct" resource type is specified in the `privateLinkResourceId`. The following are valid resource types:

[\[+\] Expand table](#)

Azure resource	Resource type	First available API version
Azure Storage	<code>Microsoft.Storage/storageAccounts</code>	<code>2020-08-01</code>
Azure Cosmos DB	<code>Microsoft.DocumentDb/databaseAccounts</code>	<code>2020-08-01</code>
Azure SQL Database	<code>Microsoft.Sql/servers</code>	<code>2020-08-01</code>
Azure Key Vault	<code>Microsoft.KeyVault/vaults</code>	<code>2020-08-01</code>
Azure Database for MySQL (preview)	<code>Microsoft.DBforMySQL/servers</code>	<code>2020-08-01-Preview</code>
Azure Functions (preview)	<code>Microsoft.Web/sites</code>	<code>2020-08-01-Preview</code>

Azure resource	Resource type	First available API version
Azure SQL Managed Instance (preview)	Microsoft.Sql/managedInstance	2020-08-01-Preview

In addition, the specified `groupId` needs to be valid for the specified resource type. For example, `groupId` "blob" is valid for type `Microsoft.Storage/storageAccounts`, it can't be used with any other resource type. For a given search management API version, customers can find out the supported `groupId` and resource type details by utilizing the [List supported API](#).

- Quota limit enforcement: Search services have quotas imposed on the distinct number of shared private link resources that can be created and the number of various target resource types that are being used (based on `groupId`). For more information, see [Shared private link resource limits](#).

Deployment failures

A search service initiates the request to create a shared private link, but Azure Resource Manager performs the actual work. You can [check the deployment's status](#) in the Azure portal or by query, and address any errors that might occur.

Shared private link resources that fail Azure Resource Manager deployment show up in [List](#) and [Get](#) API calls, but they have a "Provisioning State" of `Failed`. Once the reason of the Azure Resource Manager deployment failure is ascertained, delete the `Failed` resource and re-create it after applying the appropriate resolution from the following table.

[] [Expand table](#)

Deployment failure reason	Description	Resolution
"LinkedAuthorizationFailed"	The error message states that the client has permission to create the shared private link on the search service, but doesn't have permission to perform action 'privateEndpointConnectionApproval/action' on the linked scope.	Recheck the private link ID in the request to make sure there are no errors or omissions in the URL. If Azure AI Search and the Azure PaaS resource are in different subscriptions, and if you're using REST or a command line interface, ensure the

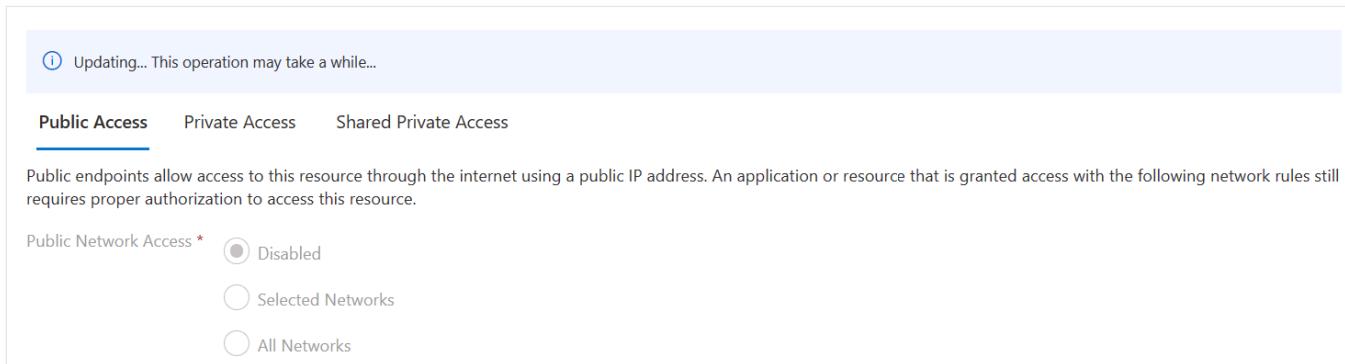
Deployment failure reason	Description	Resolution
		<p>active Azure account is the Azure PaaS resource.</p> <p>For REST clients, make sure you're not using an expired bearer token, and that the token is valid for the active subscription.</p>
Network resource provider not registered on target resource's subscription	<p>A private endpoint (and associated DNS mappings) is created for the target resource (Storage Account, Azure Cosmos DB, Azure SQL) via the <code>Microsoft.Network</code> resource provider (RP). If the subscription that hosts the target resource ("target subscription") isn't registered with <code>Microsoft.Network</code> RP, then the Azure Resource Manager deployment can fail.</p>	<p>You need to register this RP in their target subscription. You can register the resource provider using the Azure portal, PowerShell, or CLI.</p>
Invalid <code>groupId</code> for the target resource	<p>When Azure Cosmos DB accounts are created, you can specify the API type for the database account. While Azure Cosmos DB offers several different API types, Azure AI Search only supports "Sql" as the <code>groupId</code> for shared private link resources. When a shared private link of type "Sql" is created for a <code>privateLinkResourceId</code> pointing to a non-Sql database account, the Azure Resource Manager deployment fails because of the <code>groupId</code> mismatch. The Azure resource ID of an Azure Cosmos DB account isn't sufficient to determine the API type that is being used. Azure AI Search tries to create the private endpoint, which Azure Cosmos DB then denies.</p>	<p>You should ensure that the <code>privateLinkResourceId</code> of the specified Azure Cosmos DB resource is for a database account of "Sql" API type</p>
Target resource not found	<p>Existence of the target resource specified in <code>privateLinkResourceId</code> is checked only during the commencement of the Azure Resource Manager deployment. If the target resource is no longer available, then the deployment fails.</p>	<p>You should ensure that the target resource is present in the specified subscription and resource group and isn't moved or deleted.</p>
Transient/other errors	<p>The Azure Resource Manager deployment can fail if there's an infrastructure outage or because of other unexpected reasons. This should be rare and usually indicates a transient state.</p>	<p>Retry creating this resource at a later time. If the problem persists, reach out to Azure Support.</p>

Issues approving the backing private endpoint

A private endpoint is created to the target Azure resource as specified in the shared private link creation request. This is one of the final steps in the asynchronous Azure Resource Manager deployment operation, but Azure AI Search needs to link the private endpoint's private IP address as part of its network configuration. Once this link is done, the `provisioningState` of the shared private link resource goes to a terminal success state `Succeeded`. Customers should only approve or deny (or in general modify the configuration of the backing private endpoint) after the state transitions to `Succeeded`. Modifying the private endpoint in any way before this could result in an incomplete deployment operation and can cause the shared private link resource to end up (either immediately, or usually within a few hours) in a `Failed` state.

Search service network connectivity change stalled in an "Updating" state

Shared private links and private endpoints are used when search service **Public Network Access is Disabled**. Typically, changing network connectivity should succeed in a few minutes after the request is accepted. In some circumstances, Azure AI Search might take several hours to complete the connectivity change operation.



If you observe that the connectivity change operation is taking a significant amount of time, wait for a few hours. Connectivity change operations involve operations such as updating DNS records which might take longer than expected.

If **Public Network Access** is changed, existing shared private links and private endpoints might not work correctly. If existing shared private links and private endpoints stop working during a connectivity change operation, wait a few hours for the operation to complete. If they're still not working, try deleting and recreating them.

Shared private link resource stalled in an "Updating" or "Incomplete" state

Typically, a shared private link resource should go a terminal state (`Succeeded` or `Failed`) in a few minutes after the request is accepted.

In rare circumstances, Azure AI Search can fail to correctly mark the state of the shared private link resource to a terminal state (`Succeeded` or `Failed`). This usually occurs due to an unexpected failure. Shared private link resources are automatically transitioned to a `Failed` state if it's "stuck" in a nonterminal state for more than a few hours.

If the shared private link resource doesn't transition to a terminal state, wait for a few hours to ensure that it becomes `Failed` before you can delete it and re-create it. Alternatively, instead of waiting you can try to create another shared private link resource with a different name (keeping all other parameters the same).

Updating a shared private link resource

An existing shared private link resource can be updated using the [Create or Update API](#). Search only allows for narrow updates to the shared private link resource - only the request message can be modified via this API.

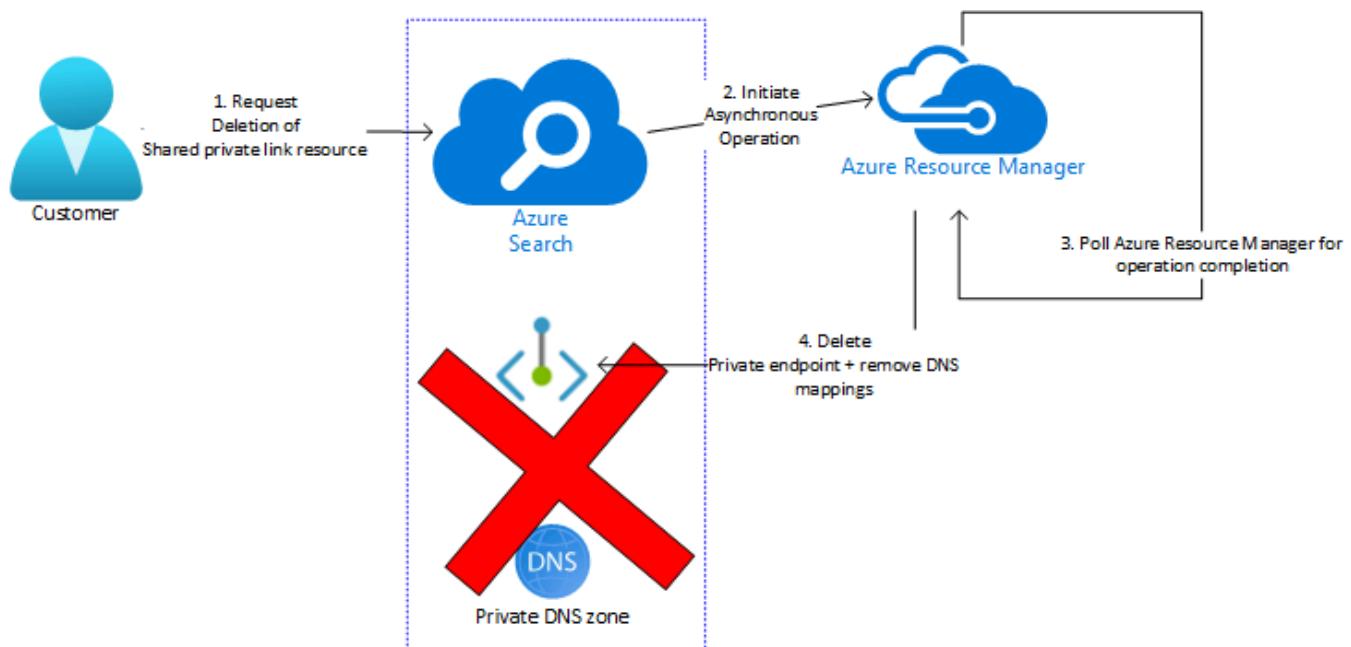
- It isn't possible to update any of the "core" properties of an existing shared private link resource (such as `privateLinkResourceId` or `groupId`) and this will always be unsupported. If any other property besides the request message needs to be changed, we advise customers to delete and re-create the shared private link resource.
- Updating the request message of a shared private link resource is only possible if it reaches the provisioning state of `Succeeded`.

Deleting a shared private link resource

Customers can delete an existing shared private link resource via the [Delete API](#). Similar to the process of creation (or update), this is also an asynchronous operation with four steps:

1. You request a search service to delete the shared private link resource.
2. The search service validates that the resource exists and is in a state valid for deletion. If so, it initiates an Azure Resource Manager delete operation to remove the resource.
3. Search queries for the completion of the operation (which usually takes a few minutes). At this point, the shared private link resource would have a provisioning state of `Deleting`.
4. Once the operation completes successfully, the backing private endpoint and any associated DNS mappings are removed. The resource doesn't show up as part of [List](#)

operation and attempting a [Get](#) operation on this resource results in a 404 Not Found.



The following are common errors that occur during the deletion phase.

 Expand table

Failure Type	Description	Resolution
Resource is in nonterminal state	A shared private link resource that's not in a terminal state (Succeeded or Failed) can't be deleted. It's possible (rare) for a shared private link resource to be stuck in a nonterminal state for up to 8 hours.	Wait until the resource reaches a terminal state and retry the delete request.
Delete operation failed with error "Conflict"	The Azure Resource Manager operation to delete a shared private link resource reaches out to the resource provider of the target resource specified in <code>privateLinkId</code> ("target RP") before it can remove the private endpoint and DNS mappings. Customers can utilize Azure resource locks to prevent any changes to their resources. When Azure Resource Manager reaches out to the target RP, it requires the target RP to modify the state of the target resource (to remove details about the private endpoint from its metadata). When the target resource has a lock configured on it (or its resource group/subscription), the Azure Resource Manager operation fails with a "Conflict" (and appropriate details). The shared private link resource won't be deleted.	Customers should remove the lock on the target resource before retrying the deletion operation. Note: This problem can also occur when customers try to delete a search service with shared private link resources that point to "locked" target resources
Delete operation failed	The asynchronous Azure Resource Manager delete operation can fail in rare cases. When this operation fails, querying the state of the asynchronous operation presents an error message and appropriate details.	Retry the operation at a later time, or reach out to Azure Support if the problem persists.

Failure Type	Description	Resolution
Resource stuck in "Deleting" state	In rare cases, a shared private link resource might be stuck in "Deleting" state for up to 8 hours, likely due to some catastrophic failure on the search RP.	Wait for 8 hours, after which the resource would transition to <code>Failed</code> state and then reissue the request.

Next steps

Learn more about shared private link resources and how to use it for secure access to protected content.

- [Make outbound connections through a shared private link](#)
- [REST API reference](#)

Add a search service to a network security perimeter

08/18/2025

A network security perimeter is a logical network boundary around your platform-as-a-service (PaaS) resources that are deployed outside of a virtual network. It establishes a perimeter for controlling public network access to resources like Azure AI Search, [Azure Storage](#), and [Azure OpenAI](#).

This article explains how to join an Azure AI Search service to a [network security perimeter](#) to control network access to your search service. By joining a network security perimeter, you can:

- Log all access to your search service in context with other Azure resources in the same perimeter.
- Block any data exfiltration from a search service to other services outside the perimeter.
- Allow access to your search service using inbound and outbound access capabilities of the network security perimeter.

You can add a search service to a network security perimeter in the Azure portal, as described in this article. Alternatively, you can use the [Azure Virtual Network Manager REST API](#) to join a search service, and use the [Search Management REST APIs](#) to view and synchronize the configuration settings.

Limitations and considerations

- For search services within a network security perimeter, indexers must use a [system or user-assigned managed identity](#) and have a role assignment that permits read-access to data sources.
- Supported indexer data sources are currently limited to [Azure Blob Storage](#), [Azure Cosmos DB for NoSQL](#), and [Azure SQL Database](#).
- Currently, within the perimeter, indexer connections to Azure PaaS for data retrieval is the primary use case. For outbound skills-driven API calls to Azure AI services, Azure OpenAI, or the Azure AI Foundry model catalog, or for inbound calls from the Azure AI Foundry for "chat with your data" scenarios you must [configure inbound and outbound rules](#) to allow the requests through the perimeter. If you require private connections for [structure-aware chunking](#) and vectorization, you should [create a shared private link](#) and a private network.

Prerequisites

- An existing network security perimeter. You can [create one to associate with your search service](#).
- [Azure AI Search](#), any billable tier, in any region.

Assign a search service to a network security perimeter

Azure Network Security Perimeter allows administrators to define a logical network isolation boundary for PaaS resources (for example, Azure Storage and Azure SQL Database) that are deployed outside virtual networks. It restricts communication to resources within the perimeter, and it allows non-perimeter public traffic through inbound and outbound access rules.

You can add Azure AI Search to a network security perimeter so that all indexing and query requests occur within the security boundary.

1. In the Azure portal, find the network security perimeter service for your subscription.
2. From the left pane, select **Settings > Associated resources**.



my-nsp



...

Network Security Perimeter

Search

◇ <<

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource visualizer

▽ Settings

Properties

Locks

Profiles

Associated resources

> Monitoring

> Automation

> Help

3. Select Add > Associate resources with an existing profile.

The screenshot shows the Azure portal interface for managing network security perimeter profiles. At the top, there are navigation links: '+ Add', 'Refresh', 'View details', 'Configure public network access', 'Move to another profile', 'Change access mode', and 'Remove'. Below these are two buttons: 'Associate resources with a new profile' (highlighted with a red box) and 'Associate resources with an existing profile'. A search bar labeled 'Search' is also present. The main area displays a table with columns: 'Associated resources' (checkbox), 'Resource type', 'Access mode', 'Profile', 'Access logs', and 'Public network acce...'. A message at the bottom states 'No Resources to display.' and 'Create or edit profiles to add resources'. A 'Create' button is located at the bottom right.

4. Select the profile you created when you created the network security perimeter for **Profile**.
5. Select **Add**, and then select your search service.

The screenshot shows the 'Select resources' dialog box. At the top, there is a search bar containing 'mysearchservice' and several filter buttons: 'Subscription equals 2 of 192 selected', 'Resource group equals all', 'Type equals 104 selected', 'Add filter', and 'More (1)'. Below the filters is a checkbox for 'Show hidden types'. The main table lists a single resource: 'mysearchservice' (Search service, Type: Search service, Resource group: MyResourceGroup, Location: West Central US, Subscription: MySubscription). The table has columns: 'Name ↑↓', 'Type ↑↓', 'Resource group ↑↓', 'Location ↑↓', and 'Subscription ↑↓'. At the bottom of the dialog are 'Select' and 'Cancel' buttons.

6. Select **Associate** in the lower-left corner to create the association.

Network security perimeter access modes

Network security perimeter supports two different access modes for associated resources:

Expand table

Mode	Description
Learning mode	This is the default access mode. In <i>learning</i> mode, network security perimeter logs all traffic to the search service that would have been denied if the perimeter was in enforced mode. This allows network administrators to understand the existing access patterns of the search service before implementing enforcement of access rules.
Enforced mode	In <i>Enforced</i> mode, network security perimeter logs and denies all traffic that isn't explicitly allowed by access rules.

Network security perimeter and search service networking settings

The `publicNetworkAccess` setting determines search service association with a network security perimeter.

- In Learning mode, the `publicNetworkAccess` setting controls public access to the resource.
- In Enforced mode, the `publicNetworkAccess` setting is overridden by the network security perimeter rules. For example, if a search service with a `publicNetworkAccess` setting of `enabled` is associated with a network security perimeter in Enforced mode, access to the search service is still controlled by network security perimeter access rules.

Change the network security perimeter access mode

1. Go to your network security perimeter resource in the Azure portal.
2. From the left pane, select **Settings > Associated resources**.



my-nsp



...

Network Security Perimeter

Search

◇ <<

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource visualizer

▽ Settings

Properties

Locks

Profiles

Associated resources

> Monitoring

> Automation

> Help

3. Find your search service in the table.

4. Select the three dots at the end of the row, and then select **Change access mode**.

The resources shown below are the members of profiles associated with this network security perimeter. Create or edit profiles to add resources.

es	Resource type	Access mode	Profile	Access logs	Public network access	Status
	microsoft.search/search...	Learning	defaultProfile	View access logs	⚠ Enabled	✓ Succeeded

No items selected

No Grouping

Configure public network access

Move to another profile

Change access mode (highlighted)

View effective configurations

Remove

5. Select your desired access mode, and then select **Apply**.

No items selected

No Grouping

es	Resource type	Access mode	Profile	Access logs	Public network access	Status
	microsoft.search/search...	Learning	defaultProfile	View access logs	⚠ Enabled	✓ Succeeded

Change access mode

Learning mode
Resource access based on NSP configuration and public network access setting

Enforced mode
Resource access based on NSP configuration and public network access setting will not apply

Apply (highlighted) **Cancel**

Enable logging network access

1. Go to your network security perimeter resource in the Azure portal.
2. From the left pane, select **Monitoring > Diagnostic settings**.



my-nsp



...

Network Security Perimeter

Search

◇ <<

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource visualizer

> Settings

▼ Monitoring

Diagnostic settings

Metrics

Logs

Workbooks

> Automation

> Help

3. Select **Add diagnostic setting**.

4. Enter any name, such as "diagnostic," for **Diagnostic setting name**.

5. Under **Logs**, select **allLogs**. **allLogs** ensures all inbound and outbound network access to resources in your network security perimeter is logged.

6. Under **Destination details**, select **Archive to a storage account** or **Send to Log Analytics workspace**. The storage account must be in the same region as the network security

perimeter. You can either use an existing storage account or create a new one. A Log Analytics workspace can be in a different region than the one used by the network security perimeter. You can also select any of the other applicable destinations.

The screenshot shows the 'Diagnostic setting' configuration page. At the top, there are buttons for Save, Discard, Delete, and Feedback. Below that, a description explains what a diagnostic setting is: 'A diagnostic setting specifies a list of categories of platform logs and/or metrics that you want to collect from a resource, and one or more destinations that you would stream them to. Normal usage charges for the destination will occur.' A link to 'Learn more about the different log categories and contents of those logs' is provided. The 'Diagnostic setting name' is set to 'diagnostic'. In the 'Logs' section, under 'Category groups', 'allLogs' is selected. Under 'Categories', several checkboxes are checked: 'Public inbound access allowed by NSP access rules.', 'Public inbound access denied by NSP access rules.', 'Public outbound access allowed by NSP access rules.', 'Public outbound access denied by NSP access rules.', 'Inbound access allowed within same perimeter.', and 'Public inbound access allowed by PaaS resource rules.'. In the 'Destination details' section, 'Send to Log Analytics workspace' is unchecked, while 'Archive to a storage account' is checked. A note states: 'You'll be charged normal data rates for storage and transactions when you send diagnostics to a storage account.' Another note says: 'Showing all storage accounts including classic storage accounts'. The 'Location' is set to 'West Central US', 'Subscription' is 'MySubscription', and 'Storage account' is 'mystorageaccount'.

7. Select **Save** to create the diagnostic setting and start logging network access.

Reading network access logs

Log Analytics workspace

The `network-security-perimeterAccessLogs` table contains all the logs for every log category (for example `network-security-perimeterPublicInboundResourceRulesAllowed`). Every log contains a record of the network security perimeter network access that matches the log category.

Here's an example of the `network-security-perimeterPublicInboundResourceRulesAllowed` log format:

[] Expand table

Column Name	Meaning	Example Value
ResultDescription	Name of the network access operation	POST /indexes/my-index/docs/search

Column Name	Meaning	Example Value
Profile	Which network security perimeter the search service was associated with	defaultProfile
ServiceResourceId	Resource ID of the search service	search-service-resource-id
Matched Rule	JSON description of the rule that was matched by the log	{ "accessRule": "IP firewall" }
SourceIPAddress	Source IP of the inbound network access, if applicable	1.1.1.1
AccessRuleVersion	Version of the network-security-perimeter access rules used to enforce the network access rules	0

Storage Account

The storage account has containers for every log category (for example `insights-logs-network-security-perimeterpublicinboundperimeterrulesallowed`). The folder structure inside the container matches the resource ID of the network security perimeter and the time the logs were taken. Each line on the JSON log file contains a record of the network security perimeter network access that matches the log category.

For example, the inbound perimeter rules allowed category log uses the following format:

```
JSON

"properties": {
    "ServiceResourceId": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/network-security-
perimeter/providers/Microsoft.Search/searchServices/network-security-perimeter-
search",
    "Profile": "defaultProfile",
    "MatchedRule": {
        "AccessRule": "myaccessrule"
    },
    "Source": {
        "IpAddress": "255.255.255.255",
    }
}
```

Add an access rule for your search service

A network security perimeter profile specifies rules that allow or deny access through the perimeter.

Within the perimeter, all resources have mutual access at the network level. You must still set up authentication and authorization, but at the network level, connection requests from inside the perimeter are accepted.

For resources outside of the network security perimeter, you must specify inbound and outbound access rules. Inbound rules specify which connections to allow in, and outbound rules specify which requests are allowed out.

A search service accepts inbound requests from apps like [Azure AI Foundry portal](#), Azure Machine Learning prompt flow, and any app that sends indexing or query requests. A search service sends outbound requests during indexer-based indexing and skillset execution. This section explains how to set up inbound and outbound access rules for Azure AI Search scenarios.

Note

Any service associated with a network security perimeter implicitly allows inbound and outbound access to any other service associated with the same network security perimeter when that access is authenticated using [managed identities and role assignments](#). Access rules only need to be created when allowing access outside of the network security perimeter, or for access authenticated using API keys.

Add an inbound access rule

Inbound access rules can allow the internet and resources outside the perimeter to connect with resources inside the perimeter.

Network security perimeter supports two types of inbound access rules:

- IP address ranges. IP addresses or ranges must be in the Classless Inter-Domain Routing (CIDR) format. An example of CIDR notation is 192.0.2.0/24, which represents the IPs that range from 192.0.2.0 to 192.0.2.255. This type of rule allows inbound requests from any IP address within the range.
- Subscriptions. This type of rule allows inbound access authenticated using any managed identity from the subscription.

To add an inbound access rule in the Azure portal:

1. Go to your network security perimeter resource in the Azure portal.
2. From the left pane, select **Settings > Profiles**.



my-nsp



...

Network Security Perimeter

Search

◇ <<

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource visualizer

▽ Settings

Properties

Locks

Profiles

Associated resources

> Monitoring

> Automation

> Help

3. Select the profile you're using with your network security perimeter.

Use profiles to associate resources and specify rules for public access to those resources. [Learn more](#)

1 item selected

Profile name	Associated resources	Policy assignments
defaultProfile	1	0

4. From the left pane, select **Settings > Inbound access rules**.

my-profile Profile

Search

Overview

Activity log

Diagnose and solve problems

Resource visualizer

Settings

- Properties
- Locks
- Inbound access rules
- Outbound access rules
- Associated resources

Automation

Help

5. Select Add.

A profile can allow external inbound access based on public IP address ranges, subscriptions or other network security perimeters. Resources associated within the same network security perimeter do not require an access rule. By default access is denied to external clients when there's no matching access rule. [Learn more](#)

<input type="checkbox"/>	Rule name	Source type	Allowed sources
No inbound access rules to display.			

Inbound access rules can allow the internet and resources outside the perimeter to connect with resources inside the perimeter.

[Add inbound access rule](#)

6. Enter or select the following values:

[Expand table](#)

Setting	Value
Rule name	The name for the inbound access rule, such as "MyInboundAccessRule."
Source type	Valid values are IP address ranges or Subscriptions .
Allowed sources	If you selected IP address ranges , enter the IP address range in CIDR format that you want to allow inbound access from. Azure IP ranges are available at this link . If you selected Subscriptions , use the subscription you want to allow inbound access from.

7. Select Add to create the inbound access rule.

Add inbound access rule

X

PREVIEW

A profile can allow external inbound access based on public IP address ranges, subscriptions or other network security perimeters. Resources associated within the same network security perimeter do not require an access rule. By default access is denied to external clients when there's no matching access rule. [Learn more](#)

Rule name *	<input type="text" value="MyRule"/> *
Source type	<input type="text" value="IP address ranges"/> ▾
Allowed sources *	<input type="text" value="1.1.1.1/32"/> *

Add

Cancel

Add an outbound access rule

A search service makes outbound calls during indexer-based indexing and skillset execution. If your indexer data sources, Azure AI services, or custom skill logic is outside of the network security perimeter, you should create an outbound access rule that allows your search service to make the connection.

Recall that in public preview, Azure AI Search can only connect to Azure Storage or Azure Cosmos DB within the security perimeter. If your indexers use other data sources, you need an outbound access rule to support that connection.

Network security perimeter supports outbound access rules based on the Fully Qualified Domain Name (FQDN) of the destination. For example, you can allow outbound access from any service associated with your network security perimeter to an FQDN such as `mystorageaccount.blob.core.windows.net`.

To add an outbound access rule in the Azure portal:

1. Go to your network security perimeter resource in the Azure portal.
2. From the left pane, select **Settings > Profiles**.



my-nsp



...

Network Security Perimeter

Search

◇ <<

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource visualizer

▽ Settings

Properties

Locks

Profiles

Associated resources

> Monitoring

> Automation

> Help

3. Select the profile you're using with your network security perimeter

Use profiles to associate resources and specify rules for public access to those resources. [Learn more](#)

1 item selected

Profile name	Associated resources	Policy assignments
defaultProfile	1	0

4. From the left pane, select **Settings > Outbound access rules**.

my-profile Profile

Search

Overview

Activity log

Diagnose and solve problems

Resource visualizer

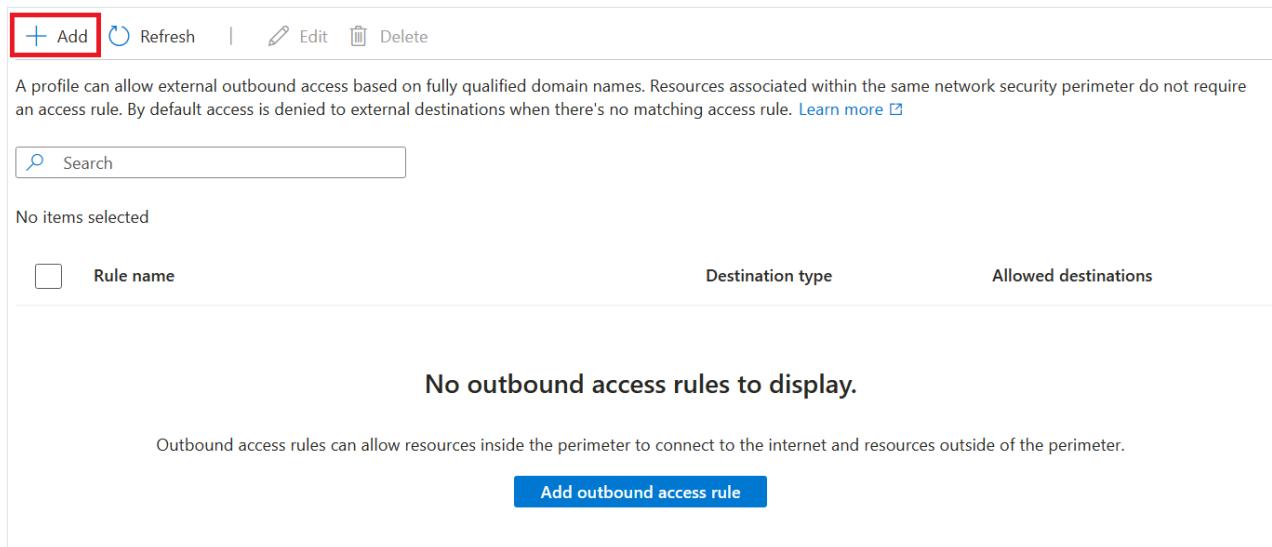
Settings

- Properties
- Locks
- Inbound access rules
- Outbound access rules**
- Associated resources

Automation

Help

5. Select Add.



A profile can allow external outbound access based on fully qualified domain names. Resources associated within the same network security perimeter do not require an access rule. By default access is denied to external destinations when there's no matching access rule. [Learn more](#)

No items selected

<input type="checkbox"/>	Rule name	Destination type	Allowed destinations
No outbound access rules to display.			

Outbound access rules can allow resources inside the perimeter to connect to the internet and resources outside of the perimeter.

[Add outbound access rule](#)

6. Enter or select the following values:

[Expand table](#)

Setting	Value
Rule name	The name for the outbound access rule, such as "MyOutboundAccessRule."
Destination type	Leave as FQDN.
Allowed destinations	Enter a comma-separated list of FQDNs you want to allow outbound access to.

7. Select Add to create the outbound access rule.

Add outbound access rule

PREVIEW

A profile can allow external outbound access based on fully qualified domain names. Resources associated within the same network security perimeter do not require an access rule. By default access is denied to external destinations when there's no matching access rule. [Learn more](#)

Rule name *	MyRule
Destination type	FQDN
Allowed destinations *	mystorageaccount.blob.core.windows.net

Add

Cancel

Test your connection through network security perimeter

In order to test your connection through network security perimeter, you need access to a web browser, either on a local computer with an internet connection or an Azure VM.

1. Change your network security perimeter association to [enforced mode](#) to start enforcing network security perimeter requirements for network access to your search service.
2. Decide if you want to use a local computer or an Azure VM.
 - a. If you're using a local computer, you need to know your public IP address.
 - b. If you're using an Azure VM, you can either use [private link](#) or [check the IP address using the Azure portal](#).
3. Using the IP address, you can create an [inbound access rule](#) for that IP address to allow access. You can skip this step if you're using private link.
4. Finally, try navigating to the search service in the Azure portal. If you can view the indexes successfully, then the network security perimeter is configured correctly.

View and manage network security perimeter configuration

You can use the [Network Security Perimeter Configuration REST APIs](#) to review and reconcile perimeter configurations.

Be sure to use `2025-05-01`, which is the latest stable REST API version. [Learn how to call the Search Management REST APIs](#).

Related content

- [Use Azure role-based access control in Azure AI Search](#)

Enable or disable role-based access control in Azure AI Search

07/31/2025

Azure AI Search supports both identity-based and [key-based authentication](#) for all data plane operations. You can use Microsoft Entra ID authentication and role-based authorization to enable identity-based access to operations and content.

Important

When you create a search service, key-based authentication is the default, but it's not the most secure option. We recommend that you replace it with role-based access as described in this article.

Before you can assign roles for authorized data plane access to Azure AI Search, you must enable role-based access control on your search service. Roles for service administration (control plane) are built in and can't be enabled or disabled.

Data plane refers to operations against the search service endpoint, such as indexing or queries, or any other operation specified in the [Search Service REST APIs](#) or equivalent Azure SDK client libraries.

Control plane refers to Azure resource management, such as creating or configuring a search service, or any other operation specified in the [Search Management REST APIs](#).

You can only enable or disable role-based access control for data plane operations. Control plane operations always use Owner, Contributor, or Reader roles. If you observe key-related activity, such as Get Admin Keys, in the [Activity Log](#) on a roles-only search service, those actions are initiated on the control plane and don't affect your content or content-related operations.

Prerequisites

- A search service in any region, on any tier, including free.
- Owner, User Access Administrator, or a custom role with [Microsoft.Authorization/roleAssignments/write](#) permissions.

Enable role-based access for data plane operations

Configure your search service to recognize an **authorization** header on data requests that provide an OAuth2 access token.

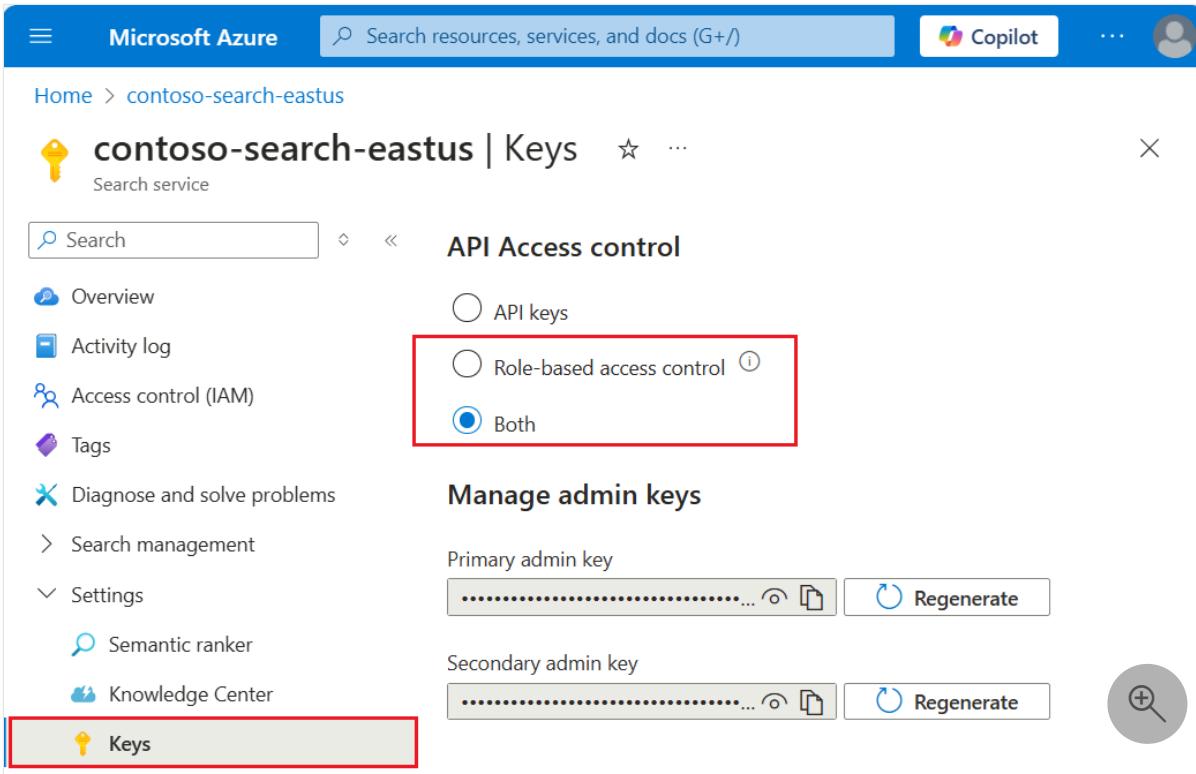
When you enable roles for the data plane, the change is effective immediately, but wait a few seconds before assigning roles.

The default failure mode for unauthorized requests is `http401WithBearerChallenge`.

Alternatively, you can set the failure mode to `http403`.

Azure portal

1. Sign in to the [Azure portal](#) and navigate to your search service.
2. Select **Settings** and then select **Keys** in the left pane.



3. Choose **Role-based control**. Only choose **Both** if you're currently using keys and need time to transition clients to role-based access control.

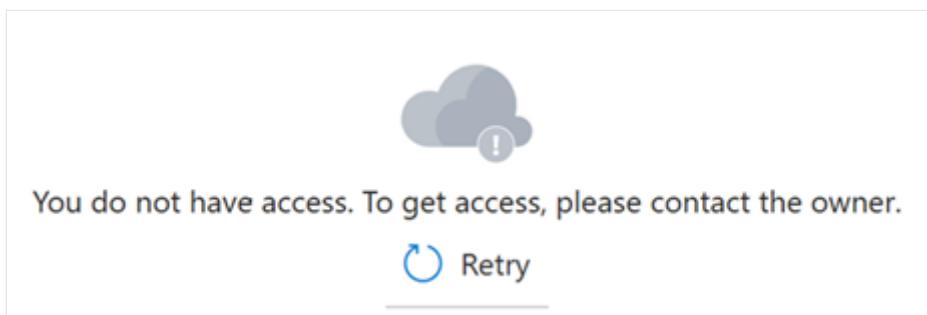
[Expand table](#)

Option	Description
API Key (default)	Requires API keys on the request header for authorization.
Role-based access control (recommended)	Requires membership in a role assignment to complete the task. It also requires an authorization header on the request.

Option	Description
Both	Requests are valid using either an API key or role-based access control, but if you provide both in the same request, the API key is used.

4. As an administrator, if you choose a roles-only approach, [assign data plane roles](#) to your user account to restore full administrative access over data plane operations in the Azure portal. Roles include Search Service Contributor, Search Index Data Contributor, and Search Index Data Reader. You need the first two roles if you want equivalent access.

Sometimes it can take five to ten minutes for role assignments to take effect. Until that happens, the following message appears in the Azure portal pages used for data plane operations.



Disable role-based access control

It's possible to disable role-based access control for data plane operations and use key-based authentication instead. You might do this as part of a test workflow, for example to rule out permission issues.

To disable role-based access control in the Azure portal:

1. Sign in to the [Azure portal](#) and open the search service page.
2. Select **Settings** and then select **Keys** in the left pane.
3. Select **API Keys**.

Disable API key authentication

[Key access](#), or local authentication, can be disabled on your service if you're exclusively using the built-in roles and Microsoft Entra authentication. Disabling API keys causes the search

service to refuse all data-related requests that pass an API key in the header.

Admin API keys can be disabled, but not deleted. Query API keys can be deleted.

Owner or Contributor permissions are required to disable security features.

Azure portal

1. In the Azure portal, navigate to your search service.
2. In the left-navigation pane, select **Keys**.
3. Select **Role-based access control**.

The change is effective immediately, but wait a few seconds before testing. Assuming you have permission to assign roles as a member of Owner, service administrator, or coadministrator, you can use portal features to test role-based access.

Next steps

[Set up roles for access to a search service](#)

Connect to Azure AI Search using roles

Azure provides a global authentication and [role-based access control](#) through Microsoft Entra ID for all services running on the platform. In this article, learn which roles provide access to search content and administration on Azure AI Search.

In Azure AI Search, you can assign Azure roles for:

- [Service administration](#)
- [Development or write-access to a search service](#)
- [Read-only access for queries](#)
- [Scoped access to a single index](#)

Per-user access over search results (sometimes referred to as *row-level security* or *document-level access*) is supported through permission inheritance for Azure Data Lake Storage (ADLS) Gen2 and Azure blob indexes and through security filters for all other platforms (see [Document-level access control](#)).

Role assignments are cumulative and pervasive across all tools and client libraries. You can assign roles using any of the [supported approaches](#) described in Azure role-based access control documentation.

Role-based access is optional, but recommended. The alternative is [key-based authentication](#), which is the default.

Prerequisites

- A search service in any region, on any tier, [enabled for role-based access](#).
- Owner, User Access Administrator, Role-based Access Control Administrator, or a custom role with [Microsoft.Authorization/roleAssignments/write](#) permissions.

How to assign roles in the Azure portal

The following steps work for all role assignments.

1. Sign in to the [Azure portal](#) ↗ .
2. Navigate to your search service.
3. [Enable role-based access](#).
4. Select **Access Control (IAM)** in the left pane.

5. Select + Add > Add role assignment to start the Add role assignment wizard.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header bar with the Microsoft Azure logo, a search bar, and a Copilot button. Below the header, the URL 'contoso-search-eastus' is shown. The main content area has a title 'contoso-search-eastus | Access control (IAM)'. On the left, there's a sidebar with links like Overview, Activity log, and Access control (IAM), which is currently selected and highlighted with a red box. The main pane has tabs for Check access, Role assignments, Roles, Deny assignments, and Classic administrators. Under the 'Check access' tab, there are sections for 'My access' (with a 'View my access' button) and 'Check access' (with a 'Check access' button). At the bottom, there are two boxes: 'Grant access to this resource' (with a 'Learn more' link) and 'View access to this resource' (with a 'Learn more' link).

6. Select a role. You can assign multiple security principals, whether users or managed identities to a role in one pass through the wizard, but you have to repeat these steps for each role you define.
7. On the **Members** tab, select the Microsoft Entra user or group identity. If you're setting up permissions for another Azure service, select a system or user-managed identity.
8. On the **Review + assign** tab, select **Review + assign** to assign the role.

Built-in roles used in search

Roles are a collection of permissions on specific operations affecting either data plane or control plane layers.

Data plane refers to operations against the search service endpoint, such as indexing or queries, or any other operation specified in the [Search Service REST APIs](#) or equivalent Azure SDK client libraries.

Control plane refers to Azure resource management, such as creating or configuring a search service.

The following roles are built in. If these roles are insufficient, [create a custom role](#).

Role	Plane	Description
Owner	Control & Data	<p>Full access to the control plane of the search resource, including the ability to assign Azure roles. Only the Owner role can enable or disable authentication options or manage roles for other users. Subscription administrators are members by default.</p> <p>On the data plane, this role has the same access as the Search Service Contributor role. It includes access to all data plane actions except the ability to query documents.</p>
Contributor	Control & Data	<p>Same level of control plane access as Owner, minus the ability to assign roles or change authentication options.</p> <p>On the data plane, this role has the same access as the Search Service Contributor role. It includes access to all data plane actions except the ability to query or index documents.</p>
Reader	Control & Data	<p>Read access across the entire service, including search metrics, content metrics (storage consumed, number of objects), and the object definitions of data plane resources (indexes, indexers, and so on). However, it can't read API keys or read content within indexes.</p>
Search Service Contributor	Control & Data	<p>Read-write access to object definitions (indexes, aliases, synonym maps, indexers, data sources, and skillsets). This role is for developers who create objects, and for administrators who manage a search service and its objects, but without access to index content. Use this role to create, delete, and list indexes, get index definitions, get service information (statistics and quotas), test analyzers, create and manage synonym maps, indexers, data sources, and skillsets. See Microsoft.Search/searchServices/* for the permissions list.</p>
Search Index Data Contributor	Data	<p>Read-write access to content in indexes. This role is for developers or index owners who need to import, refresh, or query the documents collection of an index. This role doesn't support index creation or management. By default, this role is for all indexes on a search service. See Grant access to a single index to narrow the scope.</p>
Search Index Data Reader	Data	<p>Read-only access for querying search indexes. This role is for apps and users who run queries. This role doesn't support read access to object definitions. For example, you can't read a search index definition or get search service statistics. By default, this role is for all indexes on a search service. See Grant access to a single index to narrow the scope.</p>

Combine these roles to get sufficient permissions for your use case.

(!) Note

If you disable Azure role-based access, built-in roles for the control plane (Owner, Contributor, Reader) continue to be available. Disabling role-based access removes just

the data-related permissions associated with those roles. If data plane roles are disabled, Search Service Contributor is equivalent to control-plane Contributor.

Summary

[Expand table](#)

Permissions	Search Index Reader	Search Data Contributor	Search Service Contributor	Owner/Contributor	Reader
View the resource in Azure portal	✗	✗	✓	✓	✓
View resource properties/metrics/endpoint	✗	✗	✓	✓	✓
List all objects on the resource	✗	✗	✓	✓	✓
Access quotas and service statistics	✗	✗	✓	✓	✗
Read/query an index	✓	✓	✗	✗	✗
Upload data for indexing ¹	✗	✓	✗	✗	✗
Create or edit indexes/aliases	✗	✗	✓	✓	✗
Create, edit and run indexers/data sources/skillsets	✗	✗	✓	✓	✗
Create or edit synonym maps	✗	✗	✓	✓	✗
Create or edit debug sessions	✗	✗	✓	✓	✗
Create or manage deployments	✗	✗	✓	✓	✗
Create or configure Azure AI Search resources	✗	✗	✓	✓	✗
View/Copy/Regenerate keys under Keys	✗	✗	✓	✓	✗

Permissions	Search	Search	Search	Owner/Contributor	Reader
	Index	Index Data	Service		
	Data	Contributor	Contributor		
	Reader				
View roles/policies/definitions	✗	✗	✓	✓	✗
Set authentication options	✗	✗	✓	✓	✗
Configure private connections	✗	✗	✓	✓	✗
Configure network security	✗	✗	✓	✓	✗

¹ In the Azure portal, an Owner or Contributor can run the Import data wizards that create and load indexes, even though they can't upload documents in other clients. Data connections in the wizard are made by the search service itself and not individual users. The wizards have the `Microsoft.Search/searchServices/indexes/documents/*` permission necessary for completing this task.

Owners and Contributors grant the same permissions, except that only Owners can assign roles.

Assign roles

In this section, assign roles for:

- Service administration
- Development or write-access to a search service
- Read-only access for queries

Assign roles for service administration

As a service administrator, you can create and configure a search service, and perform all control plane operations described in the [Management REST API](#) or equivalent client libraries. If you're an Owner or Contributor, you can also perform most data plane [Search REST API](#) tasks in the Azure portal.

[] [Expand table](#)

Role	ID
Owner	8e3af657-a8ff-443c-a75c-2fe8c4bcb635

Role	ID
Contributor	b24988ac-6180-42a0-ab88-20f7382dd24c
Reader	acdd72a7-3385-48ef-bd42-f606fba81ae7

Azure portal

1. Sign in to the [Azure portal](#).

2. Assign these roles:

- Owner (full access to all data plane and control plane operations, except for query permissions)
- Contributor (same as Owner, except for permissions to assign roles)
- Reader (acceptable for monitoring and viewing metrics)

Assign roles for development

Role assignments are global across the search service. To [scope permissions to a single index](#), use PowerShell or the Azure CLI to create a custom role.

[Expand table](#)

Task	Role	ID
Create or manage objects	Search Service Contributor	7ca78c08-252a-4471-8644-bb5ff32d4ba0
Load documents, run indexing jobs	Search Index Data Contributor	8ebe5a00-799e-43f5-93ac-243d3dce84a7
Query an index	Search Index Data Reader	1407120a-92aa-4202-b7e9-c0e197c71c8f

Another combination of roles that provides full access is Contributor or Owner, plus Search Index Data Reader.

Important

If you configure role-based access for a service or index and you also provide an API key on the request, the search service uses the API key to authenticate.

Azure portal

1. Sign in to the [Azure portal](#).

2. Assign these roles:

- Search Service Contributor (create-read-update-delete operations on indexes, indexers, skillsets, and other top-level objects)
- Search Index Data Contributor (load documents and run indexing jobs)
- Search Index Data Reader (query an index)

Assign roles for read-only queries

Use the Search Index Data Reader role for apps and processes that only need read-access to an index.

 [Expand table](#)

Role	ID
Search Index Data Reader with PowerShell	1407120a-92aa-4202-b7e9-c0e197c71c8f

This is a very specific role. It grants [GET or POST access](#) to the *documents collection of a search index* for search, autocomplete, and suggestions. It doesn't support GET or LIST operations on an index or other top-level objects, or GET service statistics.

This section provides basic steps for setting up the role assignment and is here for completeness, but we recommend [Use Azure AI Search without keys](#) for comprehensive instructions on configuring your app for role-based access.

Azure portal

1. Sign in to the [Azure portal](#).

2. Assign the **Search Index Data Reader** role.

Test role assignments

Use a client to test role assignments. Remember that roles are cumulative and inherited roles that are scoped to the subscription or resource group level can't be deleted or denied at the resource (search service) level.

Configure your application for keyless connections and have role assignments in place before testing.

Azure portal

1. Sign in to the [Azure portal](#).
2. Navigate to your search service.
3. On the Overview page, select the **Indexes** tab:
 - Search Service Contributors can view and create any object, but can't load documents or query an index. To verify permissions, [create a search index](#).
 - Search Index Data Contributors can load documents. There's no load documents option in the Azure portal outside of Import data wizard, but you can [reset and run an indexer](#) to confirm document load permissions.
 - Search Index Data Readers can query the index. To verify permissions, use [Search explorer](#). You should be able to send queries and view results, but you shouldn't be able to view the index definition or create one.

Test as current user

If you're already a Contributor or Owner of your search service, you can present a bearer token for your user identity for authentication to Azure AI Search.

1. Get a bearer token for the current user using the Azure CLI:

Azure CLI

```
az account get-access-token --scope https://search.azure.com/.default
```

Or by using PowerShell:

PowerShell

```
Get-AzAccessToken -ResourceUrl https://search.azure.com
```

2. Paste these variables into a new text file in Visual Studio Code.

HTTP

```
@baseUrl = PASTE-YOUR-SEARCH-SERVICE-URL-HERE  
@index-name = PASTE-YOUR-INDEX-NAME-HERE  
@token = PASTE-YOUR-TOKEN-HERE
```

3. Paste in and then send a request to confirm access. Here's one that queries the hotels-quickstart index

HTTP

```
POST https://{{baseUrl}}/indexes/{{index-name}}/docs/search?api-version=2025-09-01 HTTP/1.1  
Content-type: application/json  
Authorization: Bearer {{token}}  
  
{  
    "queryType": "simple",  
    "search": "motel",  
    "filter": "",  
    "select": "HotelName,Description,Category,Tags",  
    "count": true  
}
```

Grant access to a single index

In some scenarios, you might want to limit an application's access to a single resource, such as an index.

The Azure portal doesn't currently support role assignments at this level of granularity, but it can be done with [PowerShell](#) or the [Azure CLI](#).

In PowerShell, use [New-AzRoleAssignment](#), providing the Azure user or group name, and the scope of the assignment.

1. Load the `Azure` and `AzureAD` modules and connect to your Azure account:

PowerShell

```
Import-Module -Name Az  
Import-Module -Name AzureAD  
Connect-AzAccount
```

2. Add a role assignment scoped to an individual index:

PowerShell

```
New-AzRoleAssignment -ObjectId <objectId> `  
    -RoleDefinitionName "Search Index Data Contributor" `  
    -Scope "/subscriptions/<subscription>/resourceGroups/<resource-  
group>/providers/Microsoft.Search/searchServices/<search-  
service>/indexes/<index-name>"
```

Create a custom role

If [built-in roles](#) don't provide the right combination of permissions, you can create a [custom role](#) to support the operations you require.

This example clones **Search Index Data Reader** and then adds the ability to list indexes by name. Normally, listing the indexes on a search service is considered an administrative right.

Azure portal

These steps are derived from [Create or update Azure custom roles using the Azure portal](#). Cloning from an existing role is supported in a search service page.

These steps create a custom role that augments search query rights to include listing indexes by name. Typically, listing indexes is considered an admin function.

1. In the Azure portal, navigate to your search service.
2. In the left-navigation pane, select **Access Control (IAM)**.
3. In the action bar, select **Roles**.
4. Right-click **Search Index Data Reader** (or another role) and select **Clone** to open the **Create a custom role** wizard.
5. On the Basics tab, provide a name for the custom role, such as "Search Index Data Explorer", and then select **Next**.
6. On the Permissions tab, select **Add permission**.
7. On the Add permissions tab, search for and then select the **Microsoft Search** tile.
8. Set the permissions for your custom role. At the top of the page, using the default **Actions** selection:
 - Under Microsoft.Search/operations, select **Read : List all available operations**.
 - Under Microsoft.Search/searchServices/indexes, select **Read : Read Index**.

9. On the same page, switch to **Data actions** and under Microsoft.Search/searchServices/indexes/documents, select **Read : Read Documents**.

The JSON definition looks like the following example:

JSON

```
{  
  "properties": {  
    "roleName": "search index data explorer",  
    "description": "",  
    "assignableScopes": [  
      "/subscriptions/00000000000000000000000000000000/resourceGroups/free-search-svc/providers/Microsoft.Search/searchServices/demo-search-svc"  
    ],  
    "permissions": [  
      {  
        "actions": [  
          "Microsoft.Search/operations/read",  
          "Microsoft.Search/searchServices/indexes/read"  
        ],  
        "notActions": [],  
        "dataActions": [  
          "Microsoft.Search/searchServices/indexes/documents/read"  
        ],  
        "notDataActions": []  
      }  
    ]  
  }  
}
```

10. Select **Review + create** to create the role. You can now assign users and groups to the role.

Conditional Access

We recommend [Microsoft Entra Conditional Access](#) if you need to enforce organizational policies, such as multifactor authentication.

To enable a Conditional Access policy for Azure AI Search, follow these steps:

1. [Sign in](#) to the Azure portal.
2. Search for **Microsoft Entra Conditional Access**.
3. Select **Policies**.

4. Select **New policy**.
5. In the **Cloud apps or actions** section of the policy, add **Azure AI Search** as a cloud app depending on how you want to set up your policy.
6. Update the remaining parameters of the policy. For example, specify which users and groups this policy applies to.
7. Save the policy.

 **Important**

If your search service has a managed identity assigned to it, the specific search service will show up as a cloud app that can be included or excluded as part of the Conditional Access policy. Conditional Access policies can't be enforced on a specific search service. Instead make sure you select the general **Azure AI Search** cloud app.

Troubleshooting role-based access control issues

When developing applications that use role-based access control for authentication, some common issues might occur:

- If the authorization token came from a [managed identity](#) and the appropriate permissions were recently assigned, it [might take several hours](#) for these permissions assignments to take effect.
- The default configuration for a search service is [key-based authentication](#). If you didn't change the default key setting to **Both** or **Role-based access control**, then all requests using role-based authentication are automatically denied regardless of the underlying permissions.

Last updated on 11/21/2025

Connect your app to Azure AI Search using identities

06/17/2025

In your application code, you can set up a keyless connection to Azure AI Search that uses Microsoft Entra ID and roles for authentication and authorization. Application requests to most Azure services must be authenticated with keys or keyless connections. Developers must be diligent to never expose the keys in an unsecure location. Anyone who gains access to the key is able to authenticate to the service. Keyless authentication offers improved management and security benefits over the account key because there's no key (or connection string) to store.

Keyless connections are enabled with the following steps:

- Enable role-based access on your search service
- Set environment variables, as needed.
- Use an Azure Identity library credential type to create an Azure AI Search client object.

Prerequisites

The following steps need to be completed for both local development and production workloads:

- [Create an AI Search resource](#)
- [Enable role-based access on your search service](#)
- [Install Azure Identity client library](#)

Install Azure Identity client library

To use a keyless approach, update your AI Search enabled code with the Azure Identity client library.

.NET

Install the [Azure Identity client library for .NET](#):

.NET CLI

```
dotnet add package Azure.Identity
```

Update source code to use DefaultAzureCredential

The Azure Identity library's `DefaultAzureCredential` allows you to run the same code in the local development environment and in the Azure cloud. Create a single credential and reuse the credential instance as needed to take advantage of token caching.

.NET

For more information on `DefaultAzureCredential` for .NET, see [Azure Identity client library for .NET](#).

C#

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.Models;
using Azure.Identity;
using System;
using static System.Environment;

string endpoint = GetEnvironmentVariable("AZURE_SEARCH_ENDPOINT");
string indexName = "my-search-index";

DefaultAzureCredential credential = new();
SearchClient searchClient = new(new Uri(endpoint), indexName, credential);
SearchIndexClient searchIndexClient = new(endpoint, credential);
```

Local development

Local development using roles includes these steps:

- Assign your personal identity to RBAC roles on the specific resource.
- Use a tool like the Azure CLI or Azure PowerShell to authenticate with Azure.
- Establish environment variables for your resource.

Roles for local development

As a local developer, your Azure identity needs full control over data plane operations. These are the suggested roles:

- Search Service Contributor, create and manage objects
- Search Index Data Contributor, load and query an index

Find your personal identity with one of the following tools. Use that identity as the `<identity-id>` value.

Azure CLI

1. Sign in to Azure CLI.

```
Azure CLI
```

```
az login
```

2. Get your personal identity.

```
Azure CLI
```

```
az ad signed-in-user show \
--query id -o tsv
```

3. Assign the role-based access control (RBAC) role to the identity for the resource group.

```
Azure CLI
```

```
az role assignment create \
--role "<role-name>" \
--assignee "<identity-id>" \
--scope "/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>"
```

Where applicable, replace `<identity-id>`, `<subscription-id>`, and `<resource-group-name>` with your actual values.

Authentication for local development

Use a tool in your local development environment to authentication to Azure identity. Once you're authenticated, the `DefaultAzureCredential` instance in your source code finds and uses the authentication.

.NET

Select a tool for [authentication during local development](#).

Configure environment variables for local development

To connect to Azure AI Search, your code needs to know your resource endpoint.

Create an environment variable named `AZURE_SEARCH_ENDPOINT` for your Azure AI Search endpoint. This URL generally has the format `https://<YOUR-RESOURCE-NAME>.search.windows.net/`.

Production workloads

Deploy production workloads includes these steps:

- Choose RBAC roles that adhere to the principle of least privilege.
- Assign RBAC roles to your production identity on the specific resource.
- Set up environment variables for your resource.

Roles for production workloads

To create your production resources, you need to create a [user-assigned managed identity](#) then assign that identity to your resources with the correct roles.

The following role is suggested for a production application:

[] [Expand table](#)

Role name	Id
Search Index Data Reader	1407120a-92aa-4202-b7e9-c0e197c71c8f

Authentication for production workloads

Use the following Azure AI Search [Bicep template](#) to create the resource and set the authentication for the `identityId`. Bicep requires the role ID. The `name` shown in this Bicep snippet isn't the Azure role; it's specific to the Bicep deployment.

Bicep

```
// main.bicep
param environment string = 'production'
```

```

param roleGuid string = ''

module aiSearchRoleUser 'core/security/role.bicep' = {
    scope: aiSearchResourceGroup
    name: 'aiSearch-role-user'
    params: {
        principalId: (environment == 'development') ? principalId :
userAssignedManagedIdentity.properties.principalId
        principalType: (environment == 'development') ? 'User' :
'ServicePrincipal'
        roleDefinitionId: roleGuid
    }
}

```

The `main.bicep` file calls the following generic Bicep code to create any role. You have the option to create multiple RBAC roles, such as one for the user and another for production. This allows you to enable both development and production environments within the same Bicep deployment.

Bicep

```

// core/security/role.bicep
metadata description = 'Creates a role assignment for an identity.'
param principalId string // passed in from main.bicep

@allowed([
    'Device'
    'ForeignGroup'
    'Group'
    'ServicePrincipal'
    'User'
])
param principalType string = 'ServicePrincipal'
param roleDefinitionId string // Role ID

resource role 'Microsoft.Authorization/roleAssignments@2022-04-01' = {
    name: guid(subscription().id, resourceGroup().id, principalId,
roleDefinitionId)
    properties: {
        principalId: principalId
        principalType: principalType
        roleDefinitionId: resourceId('Microsoft.Authorization/roleDefinitions',
roleDefinitionId)
    }
}

```

Configure environment variables for production workloads

To connect to Azure AI Search, your code needs to know your resource endpoint, and the ID of the managed identity.

Create environment variables for your deployed and keyless Azure AI Search resource:

- `AZURE_SEARCH_ENDPOINT`: This URL is the access point for your Azure AI Search resource. This URL generally has the format `https://<YOUR-RESOURCE-NAME>.search.windows.net/`.
- `AZURE_CLIENT_ID`: This is the identity to authenticate as.

Related content

- [Keyless connections developer guide](#)
- [Azure built-in roles](#)
- [Set environment variables](#)

Connect to Azure AI Search using keys

Azure AI Search supports both identity-based and key-based authentication for connections to your search service. An API key is a unique string composed of 52 randomly generated numbers and letters.

In your source code, you can directly specify the API key in a request header. Alternatively, you can store it as an [environment variable](#) or app setting in your project and then reference the variable in the request.

 **Important**

When you create a search service, key-based authentication is the default, but it's not the most secure option. We recommend that you replace it with [role-based access](#).

Enabled by default

Key-based authentication is the default for new search services. A request made to a search service endpoint is accepted if both the request and the API key are valid and if the search service is configured to allow API keys on a request.

In the Azure portal, you specify authentication on the [Settings > Keys](#) page. Either **API keys** or **Both** provides key support.

Types of keys

There are two kinds of keys used for authenticating a request:

[Expand table](#)

Type	Permission level	How it's created	Maximum
Admin	Full access (read-write) for all content operations	Two admin keys, referred to as <i>primary</i> and <i>secondary</i> keys in the Azure portal, are generated when the service is created and can be individually regenerated on demand.	2 ¹
Query	Read-only access, scoped to the documents collection of a search index	One query key is generated with the service. More can be created on demand by a search service administrator.	50

¹ Having two allows you to roll over one key while using the second key for continued access to the service.

Visually, there's no distinction between an admin key or query key. Both keys are strings composed of 52 randomly generated alpha-numeric characters. If you lose track of what type of key is specified in your application, you can [check the key values in the Azure portal](#).

Key-to-role mapping

This article is about API keys. However, if you want to transition to role-based access, it's helpful to understand how keys map to [built-in roles in Azure AI Search](#):

- An admin key corresponds to the **Search Service Contributor** and **Search Index Data Contributor** roles.
- A query key corresponds to the **Search Index Data Reader** role.

Permissions to view or manage keys

Permissions for viewing and managing API keys are conveyed through [role assignments](#). Members of the following roles can view and regenerate keys:

- Owner
- Contributor
- [Search Service Contributor](#)
- Administrator and co-administrator (classic)

The following roles don't have access to API keys:

- Reader
- Search Index Data Contributor
- Search Index Data Reader

Find existing keys

You can view and manage API keys using the [Azure portal](#), [PowerShell](#), [Azure CLI](#), or [REST API](#).

Portal

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. From the left pane, select **Settings > Keys** to view admin and query keys.

The screenshot shows the Azure portal interface for managing a search service named 'my-demo-search-svc'. The left sidebar contains a navigation menu with various service management options like Semantic search, Knowledge Center, Keys, Scale, Search traffic analytics, Identity, Networking, Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic settings, and Logs. The 'Keys' option is currently selected. The main content area is titled 'API access control' and includes three radio button options: 'API keys' (selected), 'Role-based access control', and 'Both'. Below this are two sections: 'Manage admin keys' and 'Manage query keys'. The 'Manage admin keys' section shows a primary key placeholder '<your-primary-admin-key>' with a regenerate button. The 'Manage query keys' section shows a single entry with a placeholder key '<your-autogenerated-query-key>'.

Use keys on connections

API keys are used for data plane (content) requests, such as creating or accessing an index or any other request that's represented in the [Search Service REST APIs](#).

You can use either an API key or [roles](#) for control plane (service) requests. When you use an API key:

- Admin keys are used for creating, modifying, or deleting objects. Admin keys are also used to GET object definitions and system information, such as [LIST Indexes](#) or [GET Service Statistics](#).
- Query keys are typically distributed to client applications that issue queries.

REST API

Set an admin key in the request header. You can't pass admin keys on the URI or in the body of the request.

Here's an example of admin API key usage on a create index request:

HTTP

```

@baseUrl=https://my-demo-search-service.search.windows.net
@adminApiKey=aaaabbbb-0000-cccc-1111-dddd2222eeee

### Create an index
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{adminApiKey}}


{
    "name": "my-new-index",
    "fields": [
        {"name": "docId", "type": "Edm.String", "key": true, "filterable": true},
        {"name": "Name", "type": "Edm.String", "searchable": true }
    ]
}

```

Set a query key in a request header for POST, or on the URI for GET. Query keys are used for operations that target the `index/docs` collection: [Search Documents](#), [Autocomplete](#), [Suggest](#), or [GET Document](#).

Here's an example of query API key usage on a Search Documents (GET) request:

HTTP

```

### Query an index
GET /indexes/my-new-index/docs?search=*&api-version=2025-09-01&api-key={{queryApiKey}}

```

ⓘ Note

It's considered a poor security practice to pass sensitive data such as an `api-key` in the request URI. For this reason, Azure AI Search only accepts a query key as an `api-key` in the query string. As a general rule, we recommend passing your `api-key` as a request header.

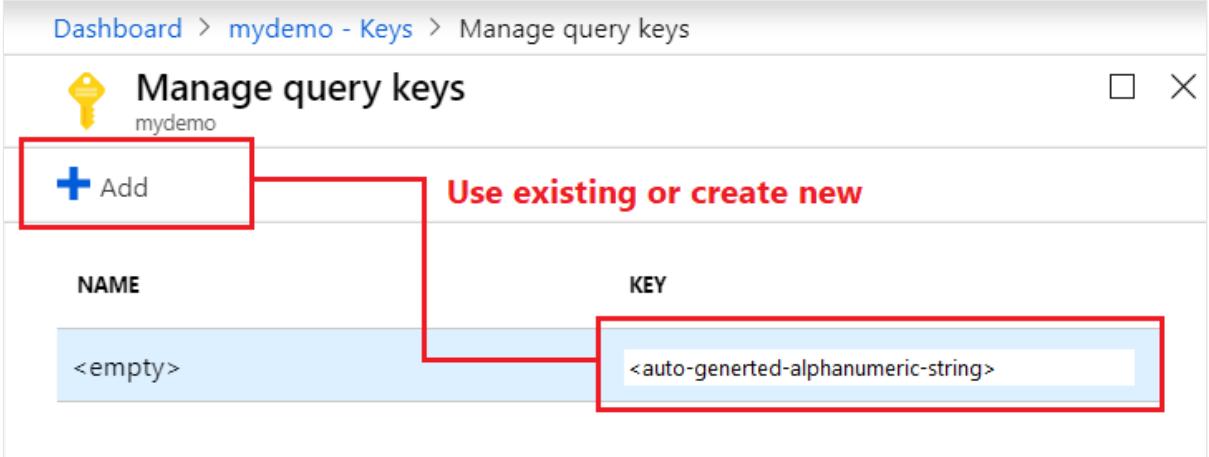
Create query keys

Query keys are used for read-only access to documents within an index for operations targeting a documents collection. Search, filter, and suggestion queries are all operations that take a query key. Any read-only operation that returns system data or object definitions, such as an index definition or indexer status, requires an admin key.

Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Portal

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. From the left pane, select **Settings > Keys** to view API keys.
3. Under **Manage query keys**, use the query key already generated for your service, or create new query keys. The default query key isn't named, but other generated query keys can be named for manageability.



Regenerate admin keys

Two admin keys are created for each service so that you can rotate a primary key while using the secondary key for business continuity.

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. From the left pane, select **Settings > Keys**.
3. Copy the secondary key.
4. For all applications, update the API key settings to use the secondary key.
5. Regenerate the primary key.
6. Update all applications to use the new primary key.

If you inadvertently regenerate both keys at the same time, all client requests using those keys will fail with HTTP 403 Forbidden. However, content isn't deleted and you aren't locked out permanently.

You can still access the service through the Azure portal or programmatically. Management functions are operative through a subscription ID not a service API key, and are thus still available even if your API keys aren't.

After you create new keys via portal or management layer, access is restored to your content (indexes, indexers, data sources, synonym maps) once you provide those keys on requests.

Secure keys

Use role assignments to restrict access to API keys.

It's not possible to use [customer-managed key encryption](#) to encrypt API keys. Only sensitive data within the search service itself (for example, index content or connection strings in data source object definitions) can be CMK-encrypted.

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. From the left pane, select **Access control (IAM)**, and then select the **Role assignments** tab.
3. In the **Role** filter, select the roles that have permission to view or manage keys (Owner, Contributor, Search Service Contributor). The resulting security principals assigned to those roles have key permissions on your search service.
4. As a precaution, also check the **Classic administrators** tab to determine whether administrators and co-administrators have access.

Best practices

- For production workloads, switch to [Microsoft Entra ID and role-based access](#). Alternatively, if you want to continue using API keys, be sure to always monitor [who has access to your API keys](#) and [regenerate API keys](#) on a regular cadence.
- Only use API keys if data disclosure isn't a risk (for example, when using sample data) and if you're operating behind a firewall. Exposing API keys puts both your data and your search service at risk of unauthorized use.
- If you use an API key, store it securely somewhere else, such as in [Azure Key Vault](#). Don't include the API key directly in your code, and never post it publicly.

- Always check code, samples, and training material before publishing to make sure you don't inadvertently expose an API key.

Related content

- [Security in Azure AI Search](#)
- [Azure role-based access control in Azure AI Search](#)
- [Manage using PowerShell](#)

Last updated on 12/01/2025

Indexer access to content protected by Azure network security

Article • 05/12/2025

If your Azure resources are deployed in an Azure virtual network, this concept article explains how a search indexer can access content that's protected by network security. It describes the outbound traffic patterns and indexer execution environments. It also covers the network protections supported by Azure AI Search and factors that might influence your security strategy. Finally, because Azure Storage is used for both data access and persistent storage, this article also covers network considerations that are specific to [search and storage connectivity](#).

Looking for step-by-step instructions instead? See [How to configure firewall rules to allow indexer access](#) or [How to make outbound connections through a private endpoint](#).

Resources accessed by indexers

Azure AI Search indexers can make outbound calls to various Azure resources in three situations:

- Connections to external data sources during indexing
- Connections to external, encapsulated code through a skillset that includes custom skills
- Connections to Azure Storage during skillset execution to cache enrichments, save debug session state, or write to a knowledge store

A list of all possible Azure resource types that an indexer might access in a typical run are listed in the table below.

 Expand table

Resource	Purpose within indexer run
Azure Storage (blobs, ADLS Gen 2, files, tables)	Data source
Azure Storage (blobs, tables)	Skillsets (caching enrichments, debug sessions, knowledge store projections)
Azure Cosmos DB (various APIs)	Data source
Azure SQL Database	Data source
OneLake (Microsoft Fabric)	Data source

Resource	Purpose within indexer run
SQL Server on Azure virtual machines	Data source
SQL Managed Instance	Data source
Azure Functions	Attached to a skillset and used to host for custom web API skills

 **Note**

An indexer also connects to Azure AI services for built-in skills. However, that connection is made over the internal network and isn't subject to any network provisions under your control.

Indexers connect to resources using the following approaches:

- A public endpoint with credentials
- A private endpoint, using Azure Private Link
- Connect as a trusted service
- Connect through IP addressing

If your Azure resource is on a virtual network, you should use either a private endpoint or IP addressing to admit indexer connections to the data.

Supported network protections

Your Azure resources could be protected using any number of the network isolation mechanisms offered by Azure. Depending on the resource and region, Azure AI Search indexers can make outbound connections through IP firewalls and private endpoints, subject to the limitations indicated in the following table.

 Expand table

Resource	IP restriction	Private endpoint
Azure Storage for text-based indexing (blobs, ADLS Gen 2, files, tables)	Supported only if the storage account and search service are in different regions.	Supported
Azure Storage for AI enrichment (caching, debug sessions, knowledge store)	Supported only if the storage account and search service are in different regions.	Supported
Azure Cosmos DB for NoSQL	Supported	Supported

Resource	IP restriction	Private endpoint
Azure Cosmos DB for MongoDB	Supported	Unsupported
Azure Cosmos DB for Apache Gremlin	Supported	Unsupported
Azure SQL Database	Supported	Supported
SQL Server on Azure virtual machines	Supported	N/A
SQL Managed Instance	Supported	N/A
Azure Functions	Supported	Supported, only for certain tiers of Azure functions

Network access and indexer execution environments

Azure AI Search has the concept of an *indexer execution environment* that optimizes processing based on the characteristics of the job. There are two environments. If you're using an IP firewall to control access to Azure resources, knowing about execution environments will help you set up an IP range that is inclusive of both environments.

For any given indexer run, Azure AI Search determines the best environment in which to run the indexer. Depending on the number and types of tasks assigned, the indexer will run in one of two environments/

[] Expand table

Execution environment	Description
Private	Internal to a search service. Indexers running in the private environment share computing resources with other indexing and query workloads on the same search service. If you set up a private connection between an indexer and your data, such as a shared private link, this is the only execution environment you can use and it's used automatically.
multitenant	Managed and secured by Microsoft at no extra cost. It isn't subject to any network provisions under your control. This environment is used to offload computationally intensive processing, leaving service-specific resources available for routine operations. Examples of resource-intensive indexer jobs include skillsets, processing large documents, or processing a high volume of documents.

Setting up IP ranges for indexer execution

This section explains IP firewall configuration for admitting requests from either execution environment.

If your Azure resource is behind a firewall, set up [inbound rules that admit indexer connections](#) for all of the IPs from which an indexer request can originate. This includes the IP address used by the search service, and the IP addresses used by the multitenant environment.

- To obtain the IP address of the search service (and the private execution environment), use `nslookup` (or `ping`) to find the fully qualified domain name (FQDN) of your search service. The FQDN of a search service in the public cloud would be `<service-name>.search.windows.net`.
- To obtain the IP addresses of the multitenant environments within which an indexer might run, use the `AzureCognitiveSearch` service tag.

[Azure service tags](#) have a published range of IP addresses of the multitenant environments for each region. You can find these IPs using the [discovery API](#) or a [downloadable JSON file](#). IP ranges are allocated by region, so check your search service region before you start.

Setting up IP rules for Azure SQL

When setting the IP rule for the multitenant environment, certain SQL data sources support a simple approach for IP address specification. Instead of enumerating all of the IP addresses in the rule, you can create a [Network Security Group rule](#) that specifies the `AzureCognitiveSearch` service tag.

You can specify the service tag if your data source is either:

- [SQL Server on Azure virtual machines](#)
- [SQL Managed Instances](#)

Notice that if you specified the service tag for the multitenant environment IP rule, you'll still need an explicit inbound rule for the private execution environment (meaning the search service itself), as obtained through `nslookup`.

Choose a connectivity approach

A search service can't be provisioned into a specific virtual network, running natively on a virtual machine. Although some Azure resources offer [virtual network service endpoints](#), this

functionality won't be offered by Azure AI Search. You should plan on implementing one of the following approaches.

[] [Expand table](#)

Approach	Details
Secure the inbound connection to your Azure resource	Configure an inbound firewall rule on your Azure resource that admits indexer requests for your data. Your firewall configuration should include the service tag for multitenant execution and the IP address of your search service. See Configure firewall rules to allow indexer access .
Private connection between Azure AI Search and your Azure resource	Configure a shared private link used exclusively by your search service for connections to your resource. Connections travel over the internal network and bypass the public internet. If your resources are fully locked down (running on a protected virtual network, or otherwise not available over a public connection), a private endpoint is your only choice. See Make outbound connections through a private endpoint .

Connections through a private endpoint must originate from the search service's private execution environment.

Configuring an IP firewall is free. A private endpoint, which is based on Azure Private Link, has a billing impact. See [Azure Private Link pricing](#) for details.

After you configure network security, follow up with role assignments that specify which users and groups have read and write access to your data and operations.

Considerations for using a private endpoint

This section narrows in on the private connection option.

- A shared private link requires a billable search service, where the minimum tier is either Basic for text-based indexing or Standard 2 (S2) for skills-based indexing. See [tier limits on the number of private endpoints](#) for details.
- Once a shared private link is created, the search service always uses it for every indexer connection to that specific Azure resource. The private connection is locked and enforced internally. You can't bypass the private connection for a public connection.
- Requires a billable Azure Private Link resource.
- Requires that a subscription owner approve the private endpoint connection.
- Requires that you turn off the multitenant execution environment for the indexer.

You do this by setting the `executionEnvironment` of the indexer to "Private". This step ensures that all indexer execution is confined to the private environment provisioned within the search service. This setting is scoped to an indexer and not the search service. If you want all indexers to connect over private endpoints, each one must have the following configuration:

JSON

```
{  
  "name" : "myindexer",  
  ... other indexer properties  
  "parameters" : {  
    ... other parameters  
    "configuration" : {  
      ... other configuration properties  
      "executionEnvironment": "Private"  
    }  
  }  
}
```

Once you have an approved private endpoint to a resource, indexers that are set to be *private* attempt to obtain access via the private link that was created and approved for the Azure resource.

Azure AI Search will validate that callers of the private endpoint have appropriate role assignments. For example, if you request a private endpoint connection to a storage account with read-only permissions, this call will be rejected.

If the private endpoint isn't approved, or if the indexer didn't use the private endpoint connection, you'll find a `transientFailure` error message in indexer execution history.

Supplement network security with token authentication

Firewalls and network security are a first step in preventing unauthorized access to data and operations. Authorization should be your next step.

We recommend role-based access, where Microsoft Entra ID users and groups are assigned to roles that determine read and write access to your service. See [Connect to Azure AI Search using role-based access controls](#) for a description of built-in roles and instructions for creating custom roles.

If you don't need key-based authentication, we recommend that you disable API keys and use role assignments exclusively.

Access to a network-protected storage account

A search service stores indexes and synonym lists. For other features that require storage, Azure AI Search takes a dependency on Azure Storage. Enrichment caching, debug sessions, and knowledge stores fall into this category. The location of each service, and any network protections in place for storage, will determine your data access strategy.

Same-region services

In Azure Storage, access through a firewall requires that the request originates from a different region. If Azure Storage and Azure AI Search are in the same region, you can bypass the IP restrictions on the storage account by accessing data under the system identity of the search service.

There are two options for supporting data access using the system identity:

- Configure search to run as a [trusted service](#) and use the [trusted service exception](#) in Azure Storage.
- Configure a [resource instance rule](#) in Azure Storage that admits inbound requests from an Azure resource.

The above options depend on Microsoft Entra ID for authentication, which means that the connection must be made with a Microsoft Entra login. Currently, only an Azure AI Search [system-assigned managed identity](#) is supported for same-region connections through a firewall.

Services in different regions

When search and storage are in different regions, you can use the previously mentioned options or set up IP rules that admit requests from your service. Depending on the workload, you might need to set up rules for multiple execution environments as described in the next section.

Next steps

Now that you're familiar with indexer data access options for solutions deployed in an Azure virtual network, review either of the following how-to articles as your next step:

- [How to make indexer connections to a private endpoint](#)
- [How to make indexer connections through an IP firewall](#)

Configure a search service to connect using a managed identity

08/27/2025

You can use Microsoft Entra ID security principals and role assignments for outbound connections from Azure AI Search to other Azure resources providing data, applied AI, or vectorization during indexing or queries.

To use roles on an outbound connection, first configure your search service to use either a [system-assigned or user-assigned managed identity](#) as the security principal for your search service in a Microsoft Entra tenant. Once you have a managed identity, you can assign roles for authorized access. Managed identities and role assignments eliminate the need for passing secrets and credentials in a connection string or code.

Prerequisites

- A search service at the [Basic tier or higher](#), any region.
- An Azure resource that accepts incoming requests from a Microsoft Entra security principal that has a valid role assignment.
- To create a managed identity, you must be an Owner or User Access Administrator roles. To assign roles, you must be an Owner, User Access Administrator, Role-based Access Control Administrator, or a member of a custom role with Microsoft.Authorization/roleAssignments/write permissions.

Supported scenarios

You can use managed identities for these scenarios.

 Expand table

Scenario	System	User-assigned
Connect to indexer data sources ¹	Yes	Yes ²
Connect to embedding and chat completion models in Azure OpenAI, Azure AI Foundry, and Azure Functions via skills/vectorizers ³	Yes	Yes
Connect to Azure Key Vault for customer-managed keys	Yes	Yes

Scenario	System	User-assigned
Connect to Debug sessions (hosted in Azure Storage) ¹	Yes	No
Connect to an enrichment cache (hosted in Azure Storage) ^{1, 4}	Yes	Yes ²
Connect to a Knowledge Store (hosted in Azure Storage) ¹	Yes	Yes ²

¹ For connectivity between search and storage, network security imposes constraints on which type of managed identity you can use. Only a system managed identity can be used for a same-region connection to Azure Storage, and that connection must be via the *trusted service exception* or resource instance rule. See [Access to a network-protected storage account](#) for details.

² User-assigned managed identities can be used in data source connection strings. However, only the newer preview REST APIs and preview packages support a user-assigned managed identity in a connection string. Be sure to switch to a preview API if you set `SearchIndexerDataUserAssignedIdentity` as the `identity` in a data source connection.

³ Connections to Azure OpenAI, Azure AI Foundry, and Azure Functions via skills/vectorizers include: [Custom skill](#), [Custom vectorizer](#), [Azure OpenAI embedding skill](#), [Azure OpenAI vectorizer](#), [AML skill](#) and [Azure AI Foundry model catalog vectorizer](#).

⁴ AI search service currently can't connect to tables on a storage account that has [shared key access turned off](#).

Create a system managed identity

A system-assigned managed identity is a Microsoft Entra ID security principal that's automatically created and linked to an Azure resource, such as an Azure AI Search service.

You can have one system-assigned managed identity for each search service. It's unique to your search service and bound to the service for its lifetime.

When you enable a system-assigned managed identity, Microsoft Entra ID creates a security principal for your search service that's used to authenticate to other Azure resources. You can then use this identity in role assignments for authorized access to data and operations.

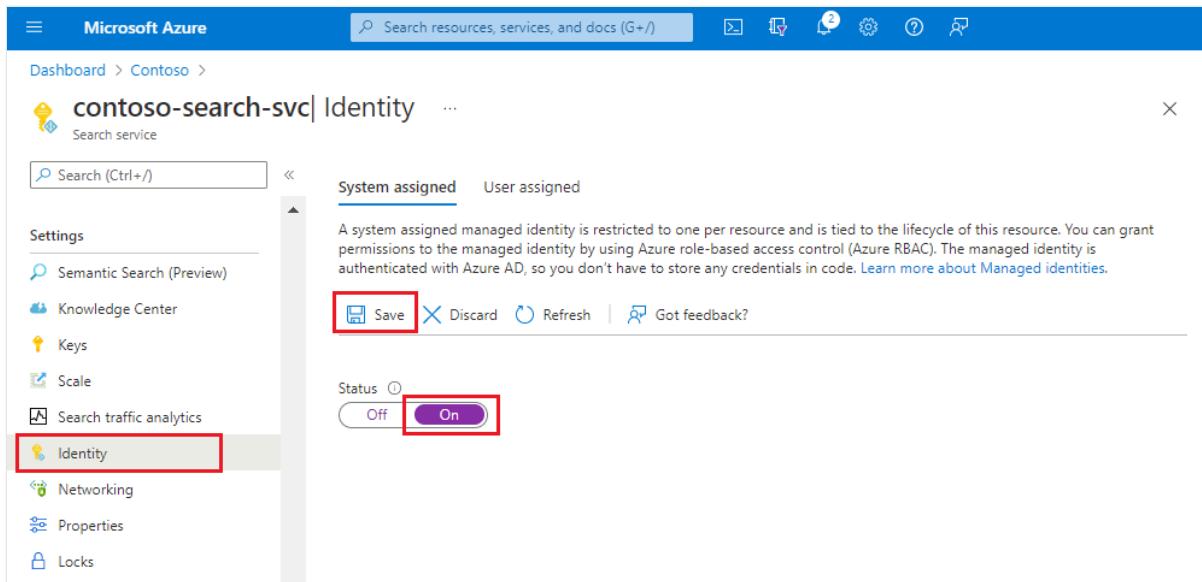
Azure portal

1. Sign in to the [Azure portal](#) and [find your search service](#).

2. From the left pane, select **Settings** > **Identity**.

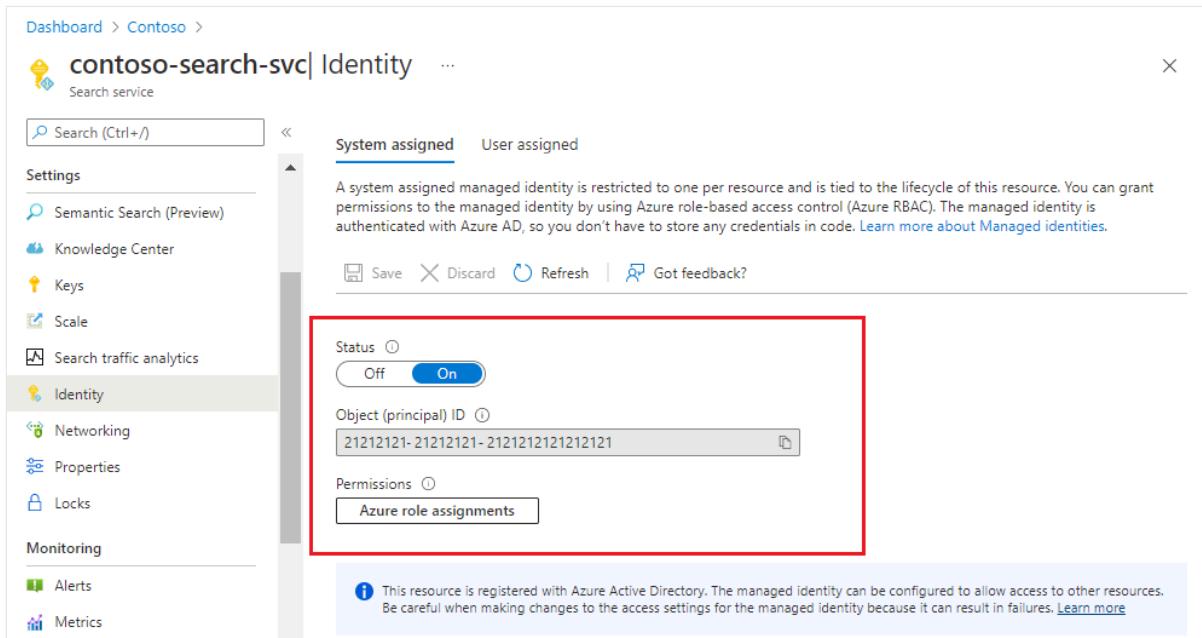
3. On the **System assigned** tab, under **Status**, select **On**.

4. Select **Save**.



The screenshot shows the Azure portal interface for managing a search service. The top navigation bar includes the Microsoft Azure logo, a search bar, and various navigation icons. Below the navigation bar, the page title is "contoso-search-svc | Identity". The left sidebar contains a "Settings" section with several options: Semantic Search (Preview), Knowledge Center, Keys, Scale, Search traffic analytics, **Identity** (which is highlighted with a red box), Networking, Properties, and Locks. The main content area is titled "System assigned". It contains a brief description of system assigned identities and a status switch. The "Status" switch is currently set to "On" (highlighted with a red box). At the bottom of the main content area are "Save", "Discard", "Refresh", and "Got feedback?" buttons.

After you save the settings, the page updates to show an object identifier that's assigned to your search service.



The screenshot shows the Azure portal interface for managing a search service. The top navigation bar and sidebar are identical to the previous screenshot. The main content area is titled "System assigned". It contains a brief description of system assigned identities and a status switch. The "Status" switch is currently set to "On". Below the status switch, there is a section labeled "Object (principal) ID" which displays the value "21212121-21212121-212121212121". There is also a section labeled "Permissions" with the sub-section "Azure role assignments". A callout message at the bottom right of the main content area states: "This resource is registered with Azure Active Directory. The managed identity can be configured to allow access to other resources. Be careful when making changes to the access settings for the managed identity because it can result in failures." The "Identity" option in the left sidebar is highlighted with a red box.

Create a user-assigned managed identity

A user-assigned managed identity is an Azure resource that can be scoped to subscriptions, resource groups, or resource types.

You can create multiple user-assigned managed identities for more granularity in role assignments. For example, you might want separate identities for different applications and scenarios. As an independently created and managed resource, it's not bound to the service itself.

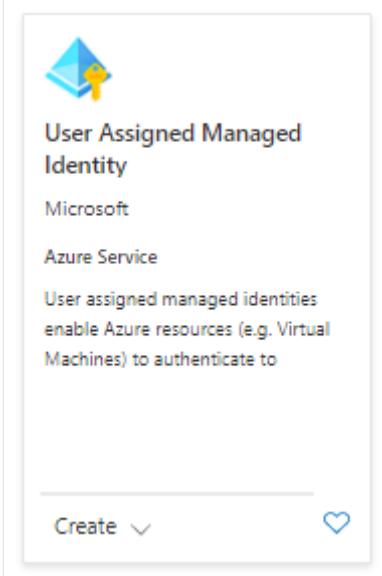
The steps for setting up a user-assigned managed identity are as follows:

- In your Azure subscription, create a user-assigned managed identity.
- On your search service, associate the user-assigned managed identity with your search service.
- On other Azure services you want to connect to, create a role assignment for the identity.

Associating a user-assigned managed identity with an Azure AI Search service is supported in the Azure portal, Search Management REST APIs, and SDK packages that provide the feature.

Azure portal

1. Sign in to the [Azure portal](#).
2. In the upper-left corner of your dashboard, select **Create a resource**.
3. Use the search box to find **User Assigned Managed Identity**, and then select **Create**.



4. Select the subscription, resource group, and region. Give the identity a descriptive name.
5. Select **Create** and wait for the resource to finish deploying.

It takes several minutes before you can use the identity.

6. On your search service page, select **Settings > Identity**.
7. On the **User assigned** tab, select **Add**.
8. Select the subscription and the user-assigned managed identity you previously created.

Assign a role

Once you have a managed identity, assign roles that determine search service permissions on the Azure resource.

- Read permissions are needed for indexer data connections and for accessing a customer-managed key in Azure Key Vault.
- Write permissions are needed for AI enrichment features that use Azure Storage for hosting debug session data, enrichment caching, and long-term content storage in a knowledge store.

The following steps illustrate the role assignment workflow. This example is for Azure OpenAI. For other Azure resources, see [Connect to Azure Storage](#), [Connect to Azure Cosmos DB](#), or [Connect to Azure SQL](#).

1. Sign in to the [Azure portal](#) with your Azure account, and go to your Azure OpenAI resource.
2. Select **Access control** from the left menu.
3. Select **Add** and then select **Add role assignment**.
4. Under **Job function roles**, select [Cognitive Services OpenAI User](#) and then select **Next**.
5. Under **Members**, select **Managed identity** and then select **Members**.
6. Filter by subscription and resource type (Search services), and then select the managed identity of your search service.
7. Select **Review + assign**.

Connection string examples

Recall from the scenarios description that you can use managed identities in connection strings to other Azure resources. This section provides examples. You can use generally available REST

API versions and Azure SDK packages for connections using a system-assigned managed identity.

💡 Tip

You can create most of these objects in the Azure portal, specifying either a system or user-assigned managed identity, and then view the JSON definition to get the connection string.

Here are some examples of connection strings for various scenarios.

Blob data source (system managed identity):

An indexer data source includes a `credentials` property that determines how the connection is made to the data source. The following example shows a connection string specifying the unique resource ID of a storage account.

A system managed identity is indicated when a connection string is the unique resource ID of a Microsoft Entra ID-aware service or application. A user-assigned managed identity is specified through an `identity` property.

JSON

```
"credentials": {  
    "connectionString": "ResourceId=/subscriptions/{subscription-  
ID}/resourceGroups/{resource-group-  
name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"  
}
```

Blob data source (user managed identity):

User-assigned managed identities can also be used in indexer data source connection strings. However, only the newer preview REST APIs and preview packages support a user-assigned managed identity in a data source connection string. Be sure to switch to a preview version if you set `SearchIndexerDataUserAssignedIdentity` as the identity in a data source connection.

A search service user identity is specified in the `identity` property.

JSON

```
"credentials": {  
    "connectionString": "ResourceId=/subscriptions/{subscription-  
ID}/resourceGroups/{resource-group-  
name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"  
},
```

```
    ...
  "identity": {
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
    "userAssignedIdentity": "/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{user-assigned-
managed-identity-name}"
  }
```

Knowledge store:

A knowledge store definition includes a connection string to Azure Storage. The connection string is the unique resource ID of your storage account. Notice that the string doesn't include containers or tables in the path. These are defined in the embedded projection definition, not the connection string.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "ResourceId=/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.Storage/storageAccounts/storage-account-name};"
}
```

Enrichment cache:

An indexer creates, uses, and remembers the container used for the cached enrichments. It's not necessary to include the container in the cache connection string. You can find the object ID on the **Identity** page of your search service in the Azure portal.

JSON

```
"cache": {
  "enableReprocessing": true,
  "storageConnectionString": "ResourceId=/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"
}
```

Debug session:

A debug session runs in the Azure portal and takes a connection string when you start the session. You can paste a string similar to the following example.

JSON

```
"ResourceId=/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name}/{container-name};",
```

Custom skill:

A [custom skill](#) targets the endpoint of an Azure function or app hosting custom code.

- `uri` is the endpoint of the function or app.
- `authResourceId` tells the search service to connect using a managed identity, passing the application ID of the target function or app in the property.

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
    "description": "A custom skill that can identify positions of different phrases  
in the source text",  
    "uri": "https://contoso.count-things.com",  
    "authResourceId": "<Azure-AD-registered-application-ID>",  
    "batchSize": 4,  
    "context": "/document",  
    "inputs": [ ... ],  
    "outputs": [ ... ]  
}
```

Connection examples for models

For connections made using managed identities, this section shows examples of connection information used by a search service to connect to a model on another resource. A connection through a system managed identity is transparent; the identity and roles are in place, and the connection succeeds if they are properly configured. In contrast, a user managed identity requires extra connection properties.

[Azure OpenAI embedding skill](#) and [Azure OpenAI vectorizer](#):

An Azure OpenAI embedding skill and vectorizer in AI Search target the endpoint of an Azure OpenAI hosting an embedding model. The endpoint is specified in the [Azure OpenAI embedding skill definition](#) and/or in the [Azure OpenAI vectorizer definition](#).

The system-managed identity is used automatically if `"apikey"` and `"authIdentity"` are empty, as demonstrated in the following example. The `"authIdentity"` property is used for user-assigned managed identity only.

System-managed identity example:

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",  
  "description": "Connects a deployed embedding model.",  
  "resourceUri": "https://url.openai.azure.com/",  
  "deploymentId": "text-embedding-ada-002",  
  "modelName": "text-embedding-ada-002",  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content"  
    }  
  ],  
  "outputs": [  
    {  
      "name": "embedding"  
    }  
  ]  
}
```

Here's a [vectorizer example](#) configured for a system-assigned managed identity. A vectorizer is specified in a search index.

JSON

```
"vectorizers": [  
  {  
    "name": "my_azure_open_ai_vectorizer",  
    "kind": "azureOpenAI",  
    "azureOpenAIParameters": {  
      "resourceUri": "https://url.openai.azure.com",  
      "deploymentId": "text-embedding-ada-002",  
      "modelName": "text-embedding-ada-002"  
    }  
  }  
]
```

User-assigned managed identity example:

A user-assigned managed identity is used if `"apiKey"` is empty and a valid `"authIdentity"` is provided.

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",  
  "description": "Connects a deployed embedding model.",
```

```

"resourceUri": "https://url.openai.azure.com/",
"deploymentId": "text-embedding-ada-002",
"modelName": "text-embedding-ada-002",
"inputs": [
  {
    "name": "text",
    "source": "/document/content"
  }
],
"outputs": [
  {
    "name": "embedding"
  }
],
"authIdentity": {
  "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
  "userAssignedIdentity":
  "/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<user-assigned-managed-identity-name>"
}
}

```

Here's a [vectorizer example](#) configured for a user-assigned managed identity. A vectorizer is specified in a search index.

JSON

```

"vectorizers": [
  {
    "name": "my_azure_open_ai_vectorizer",
    "kind": "azureOpenAI",
    "azureOpenAIParameters": {
      "resourceUri": "https://url.openai.azure.com",
      "deploymentId": "text-embedding-ada-002",
      "modelName": "text-embedding-ada-002"
      "authIdentity": {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity":
        "/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<user-assigned-managed-identity-name>"
      }
    }
  }
]

```

Check for firewall access

If your Azure resource is behind a firewall, make sure there's an inbound rule that admits requests from your search service and from the Azure portal.

- For same-region connections to Azure Blob Storage or Azure Data Lake Storage Gen2, use a system managed identity and the [trusted service exception](#). Optionally, you can configure a [resource instance rule](#) to admit requests.
- For all other resources and connections, [configure an IP firewall rule](#) that admits requests from Azure AI Search. See [Indexer access to content protected by Azure network security features](#) for details.

See also

- [Security overview](#)
- [AI enrichment overview](#)
- [Indexers overview](#)
- [Authenticate with Microsoft Entra ID](#)
- [About managed identities \(Microsoft Entra ID\)](#)

Connect to Azure Storage using a managed identity (Azure AI Search)

08/27/2025

This article explains how to configure a search service connection to an Azure Storage account using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity.

Managed identities are Microsoft Entra logins and require role assignments for access to Azure Storage.

Prerequisites

- Azure AI Search, Basic tier or higher, with a [managed identity](#).

 Note

If storage is network-protected and in the same region as your search service, you must use a system-assigned managed identity and either one of the following network options: [connect as a trusted service](#), or [connect using the resource instance rule](#).

Create a role assignment in Azure Storage

1. Sign in to Azure portal and find your storage account.
2. Select **Access control (IAM)**.
3. Select **Add** and then select **Role assignment**.
4. From the list of job function roles, select the roles needed for your search service:

 Expand table

Task	Role assignment
Blob indexing using an indexer	Add Storage Blob Data Reader
ADLS Gen2 indexing using an indexer	Add Storage Blob Data Reader

Task	Role assignment
Table indexing using an indexer	Add Storage Table Data Reader
File indexing using an indexer	Add Reader and Data Access
Write to a knowledge store	Add Storage Blob Data Contributor for object and file projections, and Reader and Data Access for table projections.
Write to an enrichment cache	Add Storage Blob Data Contributor and Storage Table Data Contributor
Save debug session state	Add Storage Blob Data Contributor

5. Select **Next**.

6. Select **Managed identity** and then select **Members**.

7. Filter by system-assigned managed identities or user-assigned managed identities. You should see the managed identity that you previously created for your search service. If you don't have one, see [Configure search to use a managed identity](#). If you already set one up but it's not available, give it a few minutes.

8. Select the identity and save the role assignment.

Specify a managed identity in a connection string

Once you have a role assignment, you can set up a connection to Azure Storage that operates under that role.

[Indexers](#) use a data source object for connections to an external data source. This section explains how to specify a system-assigned managed identity or a user-assigned managed identity on a data source connection string. You can find more [connection string examples](#) in the managed identity article.

Tip

You can create a data source connection to Azure Storage in the Azure portal, specifying either a system or user-assigned managed identity, and then view the JSON definition to see how the connection string is formulated.

System-assigned managed identity

You must have a [system-assigned managed identity already configured](#), and it must have a role-assignment on Azure Storage.

For connections made using a system-assigned managed identity, the only change to the [data source definition](#) is the format of the `credentials` property.

Provide a connection string that contains a `ResourceId`, with no account key or password. The `ResourceId` must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01

{
    "name" : "blob-datasource",
    "type" : "azureblob",
    "credentials" : {
        "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-0000-
00000000/resourceGroups/MY-DEMO-RESOURCE-
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/;"
    },
    "container" : {
        "name" : "my-container", "query" : "<optional-virtual-directory-name>"
    }
}
```

User-assigned managed identity (preview)

You must have a [user-assigned managed identity already configured](#) and associated with your search service, and the identity must have a role-assignment on Azure Storage.

Connections made through user-assigned managed identities use the same credentials as a system-assigned managed identity, plus an extra `identity` property that contains the collection of user-assigned managed identities. Only one user-assigned managed identity should be provided when creating the data source.

Provide a connection string that contains a `ResourceId`, with no account key or password. The `ResourceId` must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name.

Provide an `identity` using the syntax shown in the following example. Set `userAssignedIdentity` to the user-assigned managed identity.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-08-01-preview

{
    "name" : "blob-datasource",
    "type" : "azureblob",
    "credentials" : {
        "connectionString" : "ResourceId=/subscriptions/00000000-0000-0000-0000-00000000/resourceGroups/MY-DEMO-RESOURCE-
GROUP/providers/Microsoft.Storage/storageAccounts/MY-DEMO-STORAGE-ACCOUNT/;"
    },
    "container" : {
        "name" : "my-container", "query" : "<optional-virtual-directory-name>"
    },
    "identity" : {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity" : "/subscriptions/00000000-0000-0000-0000-00000000/resourcegroups/MY-DEMO-RESOURCE-
GROUP/providers/Microsoft.ManagedIdentity/userAssignedIdentities/MY-DEMO-USER-
MANAGED-IDENTITY"
    }
}
```

Connection information and permissions on the remote service are validated at run time during indexer execution. If the indexer is successful, the connection syntax and role assignments are valid. For more information, see [Run or reset indexers, skills, or documents](#).

Accessing network secured data in storage accounts

Azure storage accounts can be further secured using firewalls and virtual networks. If you want to index content from a storage account that is secured using a firewall or virtual network, see [Make indexer connections to Azure Storage as a trusted service](#).

See also

- [Azure blob indexer](#)
- [ADLS Gen2 indexer](#)
- [Azure table indexer](#)
- [C# Example: Index Data Lake Gen2 using Microsoft Entra ID \(GitHub\)](#) ↗

Connect to Azure Cosmos DB using a managed identity (Azure AI Search)

08/27/2025

This article explains how to set up an indexer connection to an Azure Cosmos DB database using a managed identity instead of providing credentials in the connection string.¹

You can use a system-assigned managed identity or a user-assigned managed identity. Managed identities are Microsoft Entra logins and require Azure role assignments to access data in Azure Cosmos DB. You can optionally [enforce role-based access as the only authentication method](#) for data connections by setting `disableLocalAuth` to `true` for your Azure Cosmos DB for NoSQL account.

Prerequisites

- [Create a managed identity](#) for your search service.

Supported approaches for managed identity authentication

Azure AI Search supports two mechanisms to connect to Azure Cosmos DB using managed identity.

- The *legacy* approach requires configuring the managed identity to have reader permissions on the control plane of the target Azure Cosmos DB account. Azure AI Search utilizes that identity to fetch the account keys of Cosmos DB account in the background to access the data. This approach won't work if the Cosmos DB account has `"disableLocalAuth": true`.
- The *modern* approach requires configuring the managed identity appropriate roles on the control and data plane of the target Azure Cosmos DB account. Azure AI Search will then request an access token to access the data in the Cosmos DB account. This approach works even if the Cosmos DB account has `"disableLocalAuth": true`.

Indexers that connect to Azure Cosmos DB for NoSQL support both the *legacy* and the *modern* approach - the *modern* approach is recommended.

Limitations

- Indexers that connect to Azure Cosmos DB for Gremlin and MongoDB (currently in preview) only support the *legacy* approach.

Connect to Azure Cosmos DB for NoSQL

This section outlines the steps to configure connecting to Azure Cosmos DB for NoSQL via the *modern* approach.

Configure control plane role assignments

1. Sign in to Azure portal and find your Cosmos DB for NoSQL account.
2. Select **Access control (IAM)**.
3. Select **Add** and then select **Role assignment**.
4. From the list of job function roles, select **Cosmos DB Account Reader**.
5. Select **Next**.
6. Select **Managed identity** and then select **Members**.
7. Filter by system-assigned managed identities or user-assigned managed identities. You should see the managed identity that you previously created for your search service. If you don't have one, see [Configure search to use a managed identity](#). If you already set one up but it's not available, give it a few minutes.
8. Select the identity and save the role assignment.

For more information, see [Use control plane role-based access control with Azure Cosmos DB for NoSQL](#).

Configure data plane role assignments

The managed identity needs to assigned a role to read from the Cosmos DB account's data plane. The Object (principal) ID for the search service's system/user assigned identity can be found from the search service's "Identity" tab. This step can only be performed via Azure CLI at the moment.

Set variables:

Azure CLI

```
$cosmosdb_acc_name = <cosmos db account name>
$resource_group = <resource group name>
$subsciption = <subscription ID>
$system_assigned_principal = <Object (principal) ID for the search service's
system/user assigned identity>
$readOnlyRoleDefinitionId = "00000000-0000-0000-0000-000000000001"
$scope=$(az cosmosdb show --name $cosmosdb_acc_name --resource-group
$resource_group --query id --output tsv)
```

Define a role assignment for the system-assigned identity:

Azure CLI

```
az cosmosdb sql role assignment create --account-name $cosmosdb_acc_name --
resource-group $resource_group --role-definition-id $readOnlyRoleDefinitionId --
principal-id $system_assigned_principal --scope $scope
```

For more information, see [Use data plane role-based access control with Azure Cosmos DB for NoSQL](#)

Configure the data source definition

Once you have configured **both** control plane and data plane role assignments on the Azure Cosmos DB for NoSQL account, you can set up a connection to it that operates under that role.

Indexers use a data source object for connections to an external data source. This section explains how to specify a system-assigned managed identity or a user-assigned managed identity on a data source connection string. You can find more [connection string examples](#) in the managed identity article.

Tip

You can create a data source connection to Cosmos DB in the Azure portal, specifying either a system or user-assigned managed identity, and then view the JSON definition to see how the connection string is formulated.

The [REST API](#), Azure portal, and the [.NET SDK](#) support using a system-assigned or user-assigned managed identity.

Connect through system-assigned identity

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. Provide a database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure Cosmos DB, the resource group, and the Azure Cosmos DB account name.

Here's an example using the [Create Data Source](#) REST API that exercises the *modern* approach.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
{
    "name": "my-cosmosdb-ds",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/[subscription-
id]/resourceGroups/[rg-
name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-
name];Database=[cosmos-database];IdentityAuthType=AccessToken"
    },
    "container": { "name": "[my-cosmos-collection]" }
}
```

➊ Note

If the `IdentityAuthType` property isn't part of the connection string, then Azure AI Search defaults to the *legacy* approach to ensure backward compatibility.

Connect through user-assigned identity (preview)

You need to add an "identity" property to the data source definition, where you specify the specific identity (out of several that can be assigned to the search service), that will be used to connect to the Azure Cosmos DB account.

Here's an example using user-assigned identity via the *modern* approach.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-08-01-
preview
{
    "name": "[my-cosmosdb-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/[subscription-
id]/resourceGroups/[rg-
name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-
```

```
name];Database=[cosmos-database];IdentityAuthType=AccessToken"
},
"container": { "name": "[my-cosmos-collection]"},  
"identity" : {
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
    "userAssignedIdentity": "/subscriptions/[subscription-
id]/resourcegroups/[rg-
name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[my-user-managed-
identity-name]"
}
}
```

Connect to Azure Cosmos DB for Gremlin/MongoDB (preview)

This section outlines the steps to configure connecting to Azure Cosmos DB for Gremlin/Mongo via the *legacy* approach.

Configure control plane role assignments

Follow the same steps as before to assign the appropriate roles on the control plane of the Azure Cosmos DB for Gremlin/MongoDB.

Set the connection string

- For MongoDB collections, add "ApiKind=MongoDb" to the connection string and use a preview REST API.
- For Gremlin graphs, add "ApiKind=Gremlin" to the connection string and use a preview REST API.
- For either kinds, only the **legacy** approach is supported - that is, `IdentityAuthType=AccountKey` or omitting it entirely is the only valid connection string.

Here's an example to connect to MongoDB collections using system-assigned identity via the REST API

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-08-01-
preview
{
    "name": "my-cosmosdb-ds",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/[subscription-
id]/resourceGroups/[rg-
```

```
name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-name];Database=[cosmos-database];ApiKind=MongoDb"
},
"container": { "name": "[my-cosmos-collection]", "query": null },
"dataChangeDetectionPolicy": null
}
```

Here's an example to connect to Gremlin graphs using user-assigned identity.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-08-01-preview
{
  "name": "[my-cosmosdb-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "ResourceId=/subscriptions/[subscription-id]/resourceGroups/[rg-name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-name];Database=[cosmos-database];ApiKind=Gremlin"
  },
  "container": { "name": "[my-cosmos-collection]" },
  "identity": {
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
    "userAssignedIdentity": "/subscriptions/[subscription-id]/resourcegroups/[rg-name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[my-user-managed-identity-name]"
  }
}
```

Run the indexer to verify permissions

Connection information and permissions on the remote service are validated at run time during indexer execution. If the indexer is successful, the connection syntax and role assignments are valid. For more information, see [Run or reset indexers, skills, or documents](#).

Troubleshoot connections

- For Azure Cosmos DB for NoSQL, check whether the account has its access restricted to select networks. You can rule out any firewall issues by trying the connection without restrictions in place. Refer to [Indexer access to content protected by Azure network security](#) for more information

- For Azure Cosmos DB for NoSQL, if the indexer fails due to authentication issues, ensure that the role assignments have been done **both** on the control plane and data plane of the Cosmos DB account.
- For Gremlin or MongoDB, if you recently rotated your Azure Cosmos DB account keys, you need to wait up to 15 minutes for the managed identity connection string to work.

See also

- [Indexing via an Azure Cosmos DB for NoSQL](#)
- [Indexing via an Azure Cosmos DB for MongoDB](#)
- [Indexing via an Azure Cosmos DB for Apache Gremlin](#)

Set up an indexer connection to Azure SQL using a managed identity

08/27/2025

This article explains how to set up an indexer connection to Azure SQL Database using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Microsoft Entra logins and require Azure role assignments to access data in Azure SQL.

Prerequisites

- [Create a managed identity](#) for your search service.
- [Assign an Azure admin role on SQL](#). The identity used on the indexer connection needs read permissions. You must be a Microsoft Entra admin with a server in SQL Database or SQL Managed Instance to grant read permissions on a database.
- You should be familiar with [indexer concepts](#) and [configuration](#).

1 - Assign permissions to read the database

Follow the below steps to assign the search service or user-assigned managed identity permission to read the database.

1. Connect to Visual Studio.

sql-database (contoso /sql-database)

SQL database

Search (Ctrl+ /) Copy Restore Export Set server firewall Delete Connect with... Feedback

Visual Studio

Overview Activity log Tags Diagnose and solve problems Quick start Query editor (preview)

Power Platform Power BI (preview) Power Apps (preview) Power Automate (preview)

Settings Configure Geo-Replication Connection strings Sync to other databases

Resource group (change) : Status Online Location East US Subscription (change) : Subscription ID : Tags (change) Click here to add tags

Show data for last: 1 hour 24 hours 7 days

Compute utilization

2. Authenticate with your Microsoft Entra account.

Connect

History Browse

Type here to filter the list

Local Network Azure

Server Name: sql-server-search-demo.database.windows.net

Authentication: Active Directory Interactive Authentication

User Name:

Password:

Remember Password

Database Name: sql-database

Advanced...

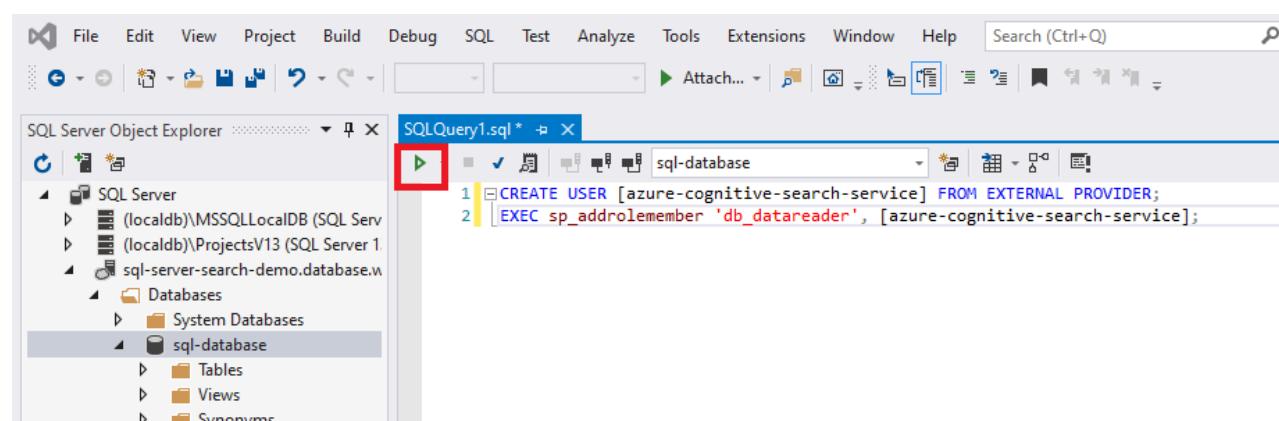
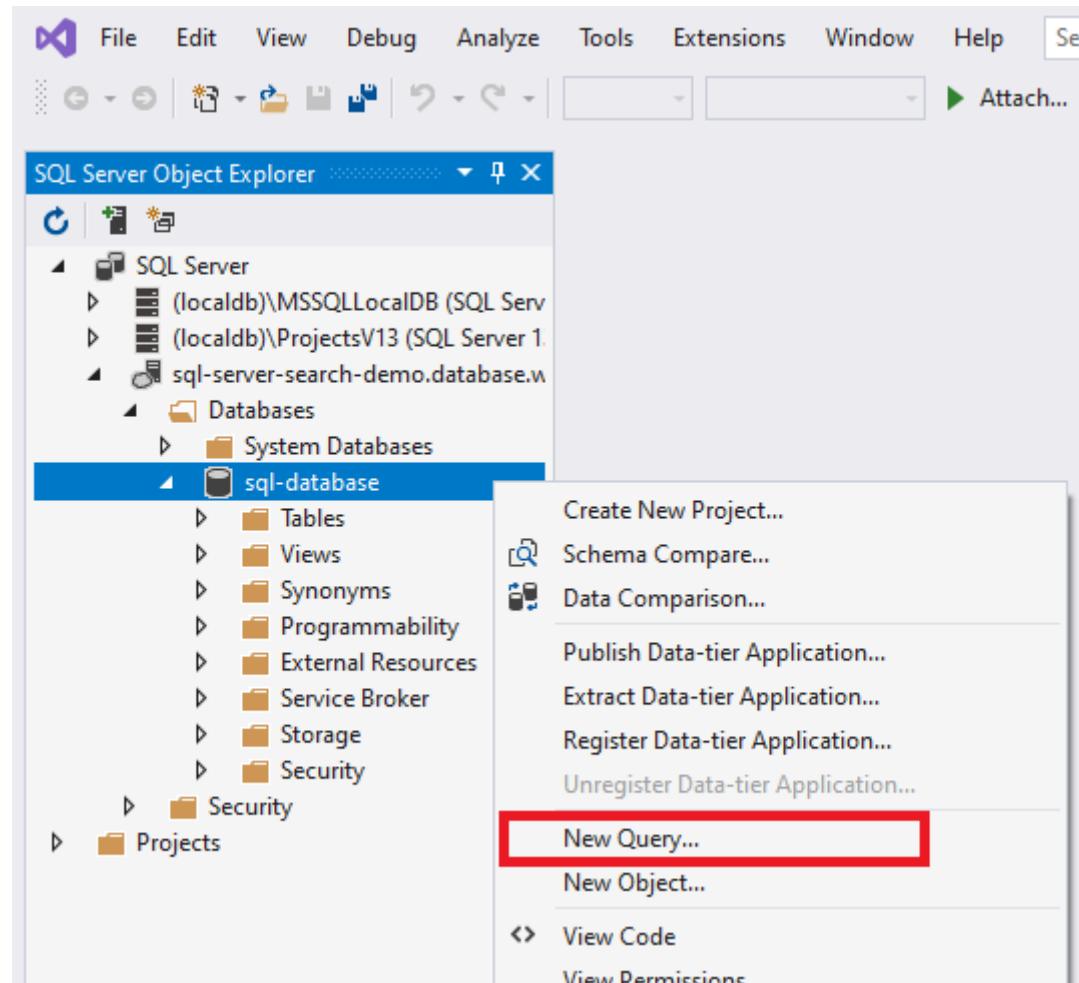
Connect Cancel

3. Execute the following commands:

Include the brackets around your search service name or user-assigned managed identity name.

```
SQL

CREATE USER [insert your search service name here or user-assigned managed identity name] FROM EXTERNAL PROVIDER;
EXEC sp_addrolemember 'db_datareader', [insert your search service name here or user-assigned managed identity name];
```



If you later change the search service identity or user-assigned identity after assigning permissions, you must remove the role membership and remove the user in the SQL database, then repeat the permission assignment. Removing the role membership and user can be accomplished by running the following commands:

SQL

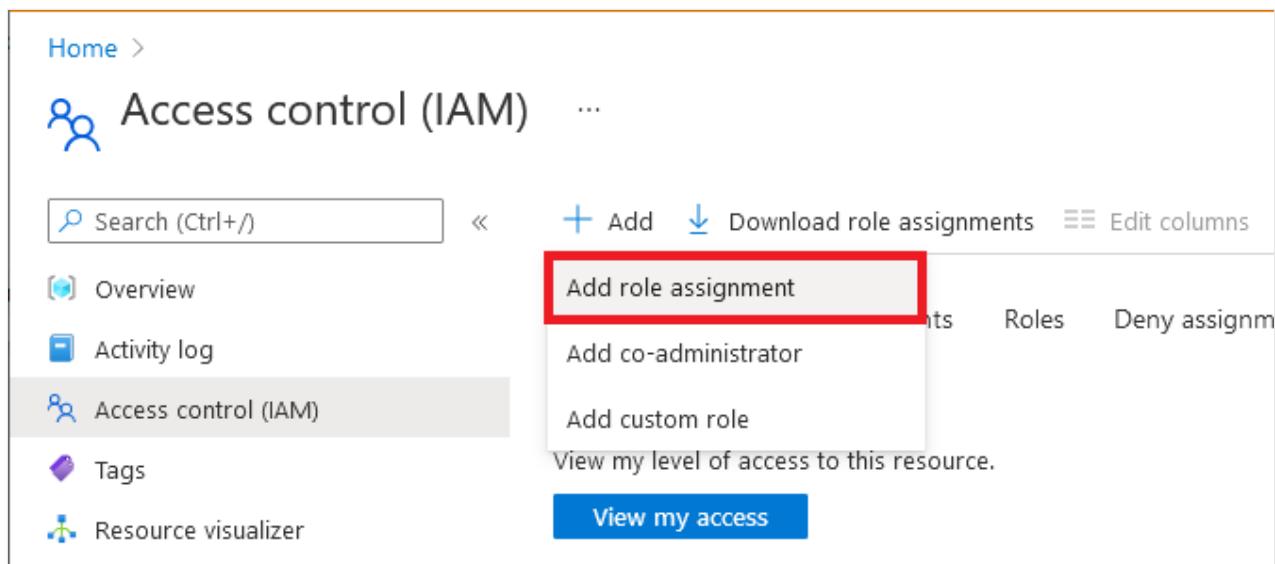
```
sp_droprolemember 'db_datareader', [insert your search service name or user-assigned managed identity name];
```

```
DROP USER IF EXISTS [insert your search service name or user-assigned managed identity name];
```

2 - Add a role assignment

In this section you'll, give your Azure AI Search service permission to read data from your SQL Server. For detailed steps, see [Assign Azure roles using the Azure portal](#).

1. In the Azure portal, navigate to your Azure SQL Server page.
2. Select **Access control (IAM)**.
3. Select **Add > Add role assignment**.



4. On the **Role** tab, select the appropriate **Reader** role.
5. On the **Members** tab, select **Managed identity**, and then select **Select members**.
6. Select your Azure subscription.
7. If you're using a system-assigned managed identity, select **System-assigned managed identity**, search for your search service, and then select it.

8. Otherwise, if you're using a user-assigned managed identity, select **User-assigned managed identity**, search for the name of the user-assigned managed identity, and then select it.
9. On the **Review + assign** tab, select **Review + assign** to assign the role.

3 - Create the data source

Create the data source and provide either a system-assigned managed identity or a user-assigned managed identity (preview).

System-assigned managed identity

The [REST API](#), Azure portal, and the Azure SDKs support system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You'll provide an Initial Catalog or Database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure SQL Database, the resource group of SQL Database, and the name of the SQL database.

Here's an example of how to create a data source to index data from a storage account using the [Create Data Source](#) REST API and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database
name];ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource group
name]/providers/Microsoft.Sql/servers/[SQL Server name];Connection Timeout=30;"
    },
    "container" : {
        "name" : "my-table"
    }
}
```

User-assigned managed identity (preview)

Preview REST APIs support connections based on a user-assigned managed identity. When you're connecting with a user-assigned managed identity, there are two changes to the data source definition:

- First, the format of the "credentials" property is an Initial Catalog or Database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure SQL Database, the resource group of SQL Database, and the name of the SQL database. This is the same format as the system-assigned managed identity.
- Second, add an "identity" property that contains the collection of user-assigned managed identities. Only one user-assigned managed identity should be provided when creating the data source. Set it to type "userAssignedIdentities".

Here's an example of how to create an indexer data source object using the most recent preview API version for [Create or Update Data Source](#):

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-08-01-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database name];ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource group name]/providers/Microsoft.Sql/servers/[SQL Server name];Connection Timeout=30;"
    },
    "container" : {
        "name" : "my-table"
    },
    "identity" : {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity" : "/subscriptions/[subscription ID]/resourcegroups/[resource group name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[managed identity name]"
    }
}
```

4 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a [Create Index](#) REST API call with a searchable `booktitle` field:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-target-index",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "booktitle", "type": "Edm.String", "searchable": true,
    "filterable": false, "sortable": false, "facetable": false }
    ]
}
```

5 - Create the indexer

An indexer connects a data source with a target search index, and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create the indexer. If the indexer is successful, the connection syntax and role assignments are valid.

Here's a [Create Indexer](#) REST API call with an Azure SQL indexer definition. The indexer runs when you submit the request.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-indexer",
    "dataSourceName" : "sql-datasource",
    "targetIndexName" : "my-target-index"
}
```

If you get an error when the indexer tries to connect to the data source that says that the client isn't allowed to access the server, take a look at [common indexer errors](#).

See also

[Azure SQL indexer](#)

Set up an indexer connection to Azure SQL Managed Instance using a managed identity

10/09/2025

This article describes how to set up an Azure AI Search indexer connection to [SQL Managed Instance](#) using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Microsoft Entra logins and require Azure role assignments to access data in SQL Managed Instance.

Before learning more about this feature, we recommended that you understand what an indexer is and how to set up an indexer for your data source. More information can be found at the following links:

- [Indexer overview](#)
- [SQL Managed Instance indexer](#)

Prerequisites

- [Create a managed identity](#) for your search service.
- Microsoft Entra admin role on SQL Managed Instance:

To assign read permissions on SQL Managed Instance, you must be an Azure Global Admin with a SQL Managed Instance. See [Configure and manage Microsoft Entra authentication with SQL Managed Instance](#) and follow the steps to provision a Microsoft Entra admin (SQL Managed Instance).

- [Configure a public endpoint and network security group in SQL Managed Instance](#) to allow connections from Azure AI Search. Connecting through a shared private link when using a managed identity isn't currently supported.

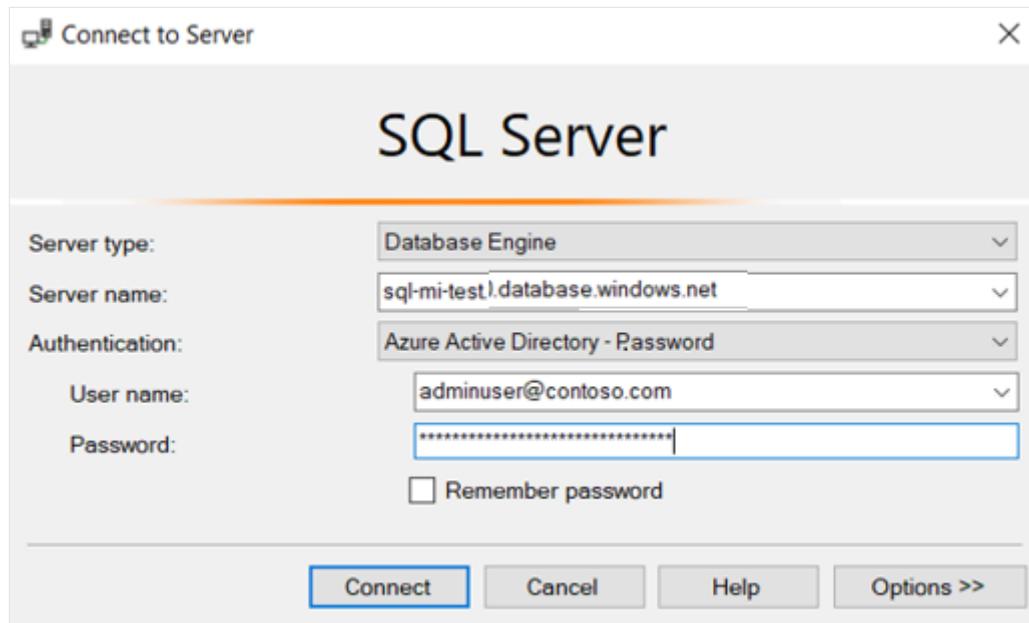
Assign permissions to read the database

Follow these steps to assign the search service system managed identity permission to read the SQL Managed database.

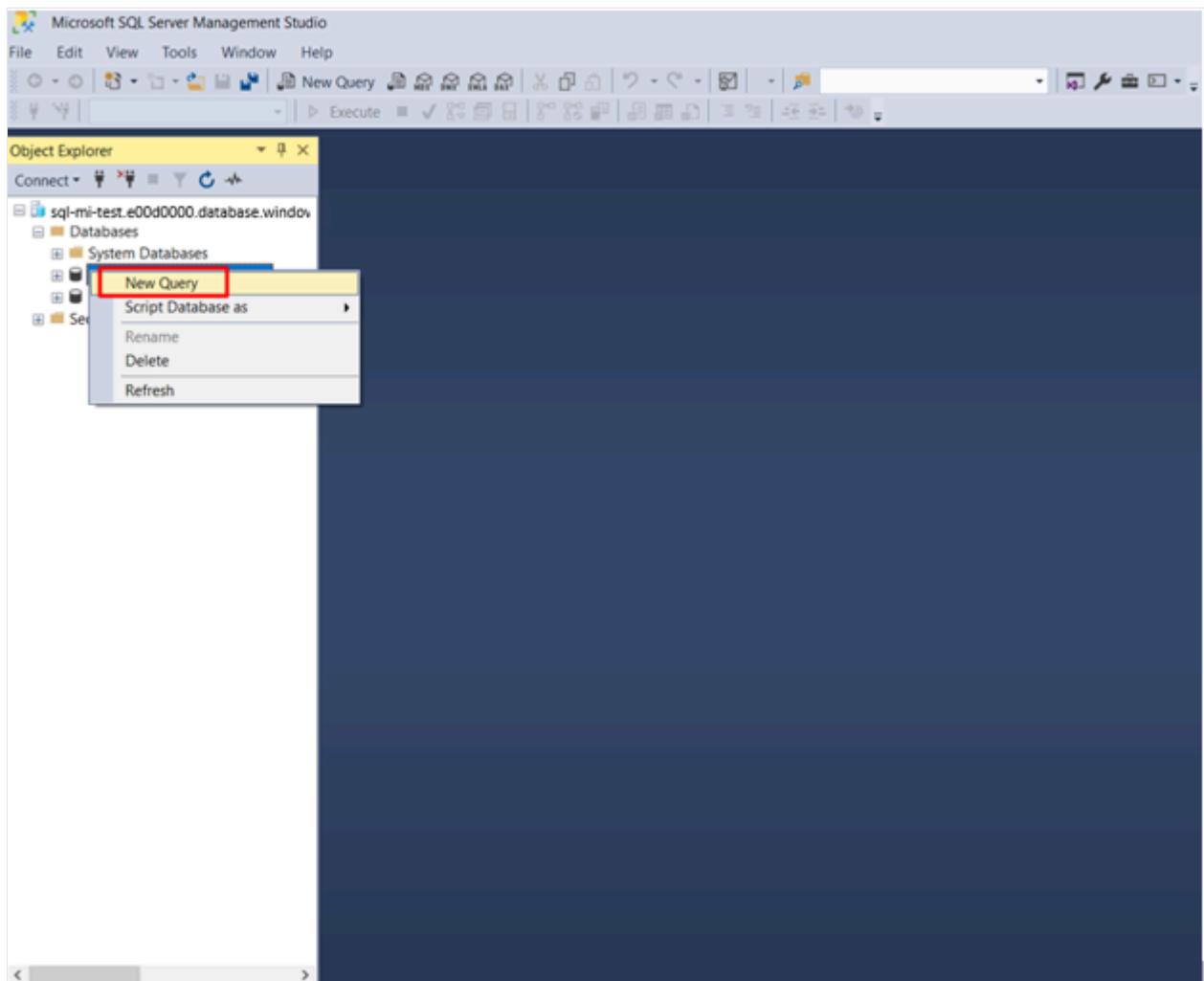
1. Connect to your SQL Managed Instance through SQL Server Management Studio (SSMS) by using one of the following methods:

- [Configure a point-to-site connection from on-premises](#)
- [Configure an Azure virtual machine](#)

2. Authenticate with your Microsoft Entra account.



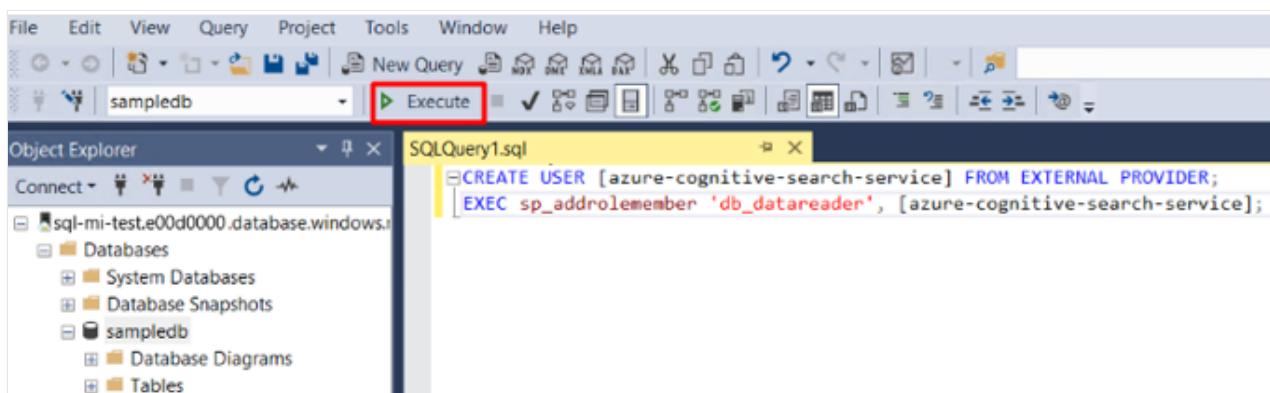
3. From the left pane, locate the SQL database you're using as data source for indexing and right-click it. Select **New Query**.



4. In the T-SQL window, copy the following commands and include the brackets around your search service name. Select **Execute**.

```
SQL

CREATE USER [insert your search service name here or user-assigned managed identity name] FROM EXTERNAL PROVIDER;
EXEC sp_addrolemember 'db_datareader', [insert your search service name here or user-assigned managed identity name];
```



If you later change the search service system identity after assigning permissions, you must remove the role membership and remove the user in the SQL database, then repeat the

permission assignment. Removing the role membership and user can be accomplished by running the following commands:

SQL

```
sp_droprolemember 'db_datareader', [insert your search service name or user-assigned managed identity name];
```

```
DROP USER IF EXISTS [insert your search service name or user-assigned managed identity name];
```

Add a role assignment

In this step, you give your Azure AI Search service permission to read data from your SQL Managed Instance.

1. In the Azure portal, navigate to your SQL Managed Instance page.
2. Select **Access control (IAM)**.
3. Select **Add** then **Add role assignment**.

The screenshot shows the Azure portal's Access control (IAM) blade for a SQL Managed Instance named Contoso. On the left, there's a sidebar with various navigation options like Overview, Activity log, and Resource Management. The main area shows the 'Add role assignment' button highlighted with a red box. Below it, there are sections for 'My access' (with a 'View my access' button), 'Check access' (with a 'Find' dropdown set to 'User, group, or service principal'), and 'Grant access to this resource' (with a 'Add role assignment' button). There's also a 'View deny assignments' section with a 'View' button. A search bar at the bottom allows searching by name or email address.

4. Select **Reader** role.

5. Leave **Assign access to** as Microsoft Entra user, group, or service principal.

6. If you're using a system-assigned managed identity, search for your search service, then select it. If you're using a user-assigned managed identity, search for the name of the user-assigned managed identity, then select it. Select **Save**.

Example for SQL Managed Instance using a system-assigned managed identity:

Home > sql-mi-test >

Add role assignment ...

Got feedback?

Role Members Review + assign

Selected role Owner

Assign access to User, group, or service principal Managed identity

Members + Select members

Name	Object ID	Type
azure-cognitive-search-service	000000-0000-0000-000000000000	Search service

Description

Review + assign Previous Next

Create the data source

Create the data source and provide a system-assigned managed identity.

System-assigned managed identity

The [REST API](#), Azure portal, and the [.NET SDK](#) support system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You provide an Initial Catalog or Database name and a `ResourceId` that has no account key or password. The `ResourceId` must include the subscription ID of SQL Managed Instance, the resource group of SQL Managed instance, and the name of the SQL database.

Here's an example of how to create a data source to index data from a storage account using the [Create Data Source REST API](#) and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-mi-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database
name];ResourceId=/subscriptions/[subscription ID]/resourcegroups/[resource group
name]/providers/Microsoft.Sql/managedInstances/[SQL Managed Instance
name];Connection Timeout=100;"
    },
    "container" : {
        "name" : "my-table"
    }
}
```

Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a [Create Index REST API](#) call with a searchable `booktitle` field:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-target-index",
    "fields": [
```

```
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "booktitle", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false }
    ]
}
```

Create the indexer

An indexer connects a data source with a target search index, and provides a schedule to automate the data refresh. Once the index and data source are created, you're ready to create the indexer.

Here's a [Create Indexer](#) REST API call with an Azure SQL indexer definition. The indexer runs when you submit the request.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2025-09-01
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-mi-indexer",
    "dataSourceName" : "sql-mi-datasource",
    "targetIndexName" : "my-target-index"
}
```

Troubleshooting

If you get an error when the indexer tries to connect to the data source that says that the client isn't allowed to access the server, see the [common indexer errors](#).

You can also rule out any firewall issues by trying the connection with and without restrictions in place.

See also

- [SQL Managed Instance and Azure SQL Database indexer](#)

Authenticate to an Azure Function App using "Easy Auth" (Azure AI Search)

This article explains how to set up an indexer connection to an Azure Function app using the [built-in authentication capabilities of Azure App Service](#), also known as "Easy Auth." Azure Function apps are a great solution for hosting Custom Web APIs that an Azure AI Search service can use to enrich content ingested during an indexer run or, if you're using a custom embedding model for [integrated vectorization](#), vectorize content in a search query.

You can use a system-assigned or user-assigned managed identity of the search service to authenticate against the Azure Function app. This approach requires setting up a Microsoft Entra ID application registration to use as the authentication provider for the Azure Function app, which is explained in this article.

Prerequisites

- A [managed identity](#) for your search service.

Configure Microsoft Entra ID application as the authentication provider

To use Microsoft Entra ID as an authentication provider to the Azure Function app, an application registration must be created. There are two options: create one automatically via the Azure Function app itself or use an existing application. To learn more about these steps, see the [App Service documentation](#).

Regardless of the option, ensure that the app registration is configured per the following steps to ensure it's compatible with Azure AI Search.

Ensure the app registration has application ID URI configured

The app registration should be configured with an application ID URI, which can be used as the token audience with Azure Function apps and Azure AI Search. Configure it in the format `api://<applicationId>`. This can be done by navigating to the [Overview](#) section of the app registration and setting the [Application ID URI](#) field.



[Screenshot of an app registration configured with application ID URI.](#)

Set supported account types for authentication

Navigate to the **Authentication** section of the app registration and configure the **supported account types** so that only accounts in the same organization directory as the app registration can utilize it for authentication.



[Screenshot of an app registration with supported account types configured.](#)

(Optional) Configure a client secret

App Service recommends using a client secret for the authentication provider application. Authentication still works without client secret, as long as the delegated permissions are set up. To set up a client secret, navigate to the **Certificates & secrets** section of the app registration, and add a **New client secret** as explained [in this article](#).

The screenshot shows the Azure portal interface for managing an app registration named 'contoso-app-registration'. The left sidebar lists various management sections like Overview, Quickstart, Integration assistant, Diagnose and solve problems, Manage, Branding & properties, Authentication, Certificates & secrets (which is currently selected), Token configuration, API permissions, Expose an API, App roles, Owners, Roles and administrators, and Manifest. The right pane displays the 'Certificates & secrets' blade. It shows three tabs: Certificates (0), Client secrets (0, which is underlined in blue), and Federated credentials (0). Below the tabs, a note states: 'A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.' A table lists one client secret entry: Description: 'New client secret', Expires: 'Never', Value: '(redacted)', and Secret ID: 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX'. A note at the bottom says: 'No client secrets have been created for this application.'

Add a scope to delegate permissions

Navigate to the section **Expose an API** and configure the app registration to have a scope that delegates admin and user permissions to it, to ensure that it's compatible with the indexer's authentication flow.

The screenshot shows the Azure portal interface for managing an application registration. On the left, a sidebar lists various management options like Overview, Quickstart, Integration assistant, Diagnose and solve problems, Manage, Branding & properties, Authentication, Certificates & secrets, Token configuration, API permissions, and Expose an API. The 'Expose an API' option is currently selected. The main content area displays the 'Application ID URI' as 'api://00000000-0000-0000-0000-000000000000'. A red box highlights the 'Scopes defined by this API' section, which contains a table with one row. The table columns are 'Scopes', 'Who can consent', 'Admin consent display name', and 'User consent display na...'. The single scope listed is 'api://00000000-0000-0000-0000-000000000000/user_impersonation', with 'Admins and users' under 'Who can consent' and 'Accesscontoso-app-registration' under 'Admin consent display name'. Below this table, there's a section for 'Authorized client applications' with a note about trusting the application and a link to 'Add a client application'. A search icon is located in the bottom right corner of the main content area.

Once the delegated permissions scope is set up, you should notice in the **API permissions** section of the app registration that the **User.Read** API on Microsoft.Graph is set.

Screenshot of an app registration with delegated permissions.

Configure Microsoft Entra ID authentication provider in the Azure Function app

With the client application registered with the previous specifications, Microsoft Entra ID authentication for the Azure Function app can be set up by following the [App Service documentation](#). Navigate to the **Authentication** section of the Azure Function app to set up the authentication details.

Ensure the following settings are configured to ensure that Azure AI Search can successfully authenticate to the Azure Function app.

Configure authentication settings

- Ensure that **App Service authentication** is **Enabled**
- Restrict access to the Azure Function app to **Require authentication**
- For **Unauthenticated requests** prefer **HTTP 401: Unauthorized**

The following screenshot highlights these specific settings for a sample Azure Function app.

Screenshot of an Azure Function app that has configured authentication settings.

Add Microsoft Entra ID authentication provider

- Add Microsoft Entra ID as the authentication provider for the Azure Function app.
- Either create a new app registration or choose a previously configured app registration. Ensure that it's configured according to the guidelines in the previous section of this document.
- Ensure that in the **Allowed token audiences** section, the application ID URI of the app registration is specified. It should be in the `api://<applicationId>` format, matching what was configured with the app registration created earlier.
- If you desire, you can configure other checks to restrict access specifically to the indexer.



Screenshot of an Azure Function app with Microsoft Entra ID Authentication provider.

Configure other checks

- Ensure that the **Object (principal) ID** of the specific Azure AI Search service's identity is specified as the **Identity requirement**, by checking the option **Allow requests from specific identities** and entering the **Object (principal) ID** in the identity section.



Screenshot of the identity section for an Azure AI Search service.

- In **Client application requirement**, select the option **Allow requests from specific client application**. You need to look up the Client ID for the Azure AI Search service's identity. To do this, copy over the Object (principal) ID from the previous step and look up in your Microsoft Entra ID tenant. There should be a matching enterprise application whose overview page lists an **Application ID**, which is the GUID that needs to be specified as the client application requirement.



Screenshot of the enterprise application details of an Azure AI Search service's identity.

Note

This step is the most important configuration on the Azure Function app and doing it wrongly can result in the indexer being forbidden from accessing the Azure Function app. Ensure that you perform the lookup of the identity's enterprise application details correctly, and you specify the **Application ID** and **Object (principal) ID** in the right places.

- For the **Tenant requirement**, choose any of the options that aligns with your security posture. For more information, see the [App Service documentation](#).

Set up a connection to the Azure Function app

Depending on whether the connection to the Azure Function app needs to be made in a Custom Web API skill or a Custom Web API vectorizer, the JSON definition is slightly different. In both cases, ensure that you specify the correct URI to the Azure Function app and set the `authResourceId` to be the same value as the **Allowed token audience** configured for the authentication provider.

Depending on whether you choose to connect using a system-assigned identity or user-assigned identity, required properties differ slightly.

Use a system-assigned identity

Here's an example to call into a function named `test` for the sample Azure Function app, where the system assigned identity of the search service is allowed to authenticate via "Easy Auth".

JSON

```
"uri": "https://contoso-function-app.azurewebsites.net/api/test?",  
"authResourceId": "api://00000000-0000-0000-0000-000000000000"
```

Use a user-assigned identity

Here's an example to call into the same function, where the specific user assigned identity is allowed to authenticate via "Easy Auth". You're expected to specify the resource ID of the exact user assigned identity to use in the `identity` property of the configuration.

JSON

```
"uri": "https://contoso-function-app.azurewebsites.net/api/test?",  
"authResourceId": "api://00000000-0000-0000-0000-000000000000",  
"identity" : {  
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",  
    "userAssignedIdentity": "/subscriptions/[subscription-  
id]/resourcegroups/[rg-  
name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[my-user-managed-  
identity-name]"  
}
```

! Note

This user assigned identity should actually be assigned to the Azure AI Search service for it to be specified in the Custom Web skill/vectorizer definition.

Run the indexer/vectorizer to verify permissions

For Custom Web API skills, permissions are validated during indexer run-time. For vectorizer, they're validated when a vector query is issued utilizing the Custom Web API vectorizer. To rule out any specific issues with authentication, you can test by disabling the authentication provider on the Azure Function app and ensuring that calls from indexer/vectorizer succeed.

- If authentication issues persist, ensure that the right identity information - namely Application ID, Object (principal) ID for the Azure AI Search service's identity is specified in the Azure Function app's authentication provider.

Related content

- [Custom Web API skill](#)
- [Custom Web API vectorizer](#)

Last updated on 11/21/2025

Configure IP firewall rules to allow indexer connections from Azure AI Search

05/29/2025

On behalf of an indexer, a search service issues outbound calls to an external Azure resource to pull in data during indexing. If your Azure resource uses IP firewall rules to filter incoming calls, you must create an inbound rule in your firewall that admits indexer requests.

This article explains how to find the IP address of your search service and configure an inbound IP rule on an Azure Storage account. While specific to Azure Storage, this approach also works for other Azure resources that use IP firewall rules for data access, such as Azure Cosmos DB and Azure SQL.

! Note

Applicable to Azure Storage only. Your storage account and your search service must be in different regions if you want to define IP firewall rules. If your setup doesn't permit this, try the [trusted service exception](#) or [resource instance rule](#) instead.

For private connections from indexers to any supported Azure resource, we recommend setting up a [shared private link](#). Private connections travel the Microsoft backbone network, bypassing the public internet completely.

Get a search service IP address

1. Get the fully qualified domain name (FQDN) of your search service. This looks like `<search-service-name>.search.windows.net`. You can find the FQDN by looking up your search service on the Azure portal.

The screenshot shows the Azure portal interface for a search service named 'contoso-search-svc'. The left sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Semantic Search (Preview), Knowledge Center, Keys, Scale, Search traffic analytics, Identity, and Properties. The 'Overview' tab is currently selected. In the main content area, there's a 'Create a Standard search service for scalability and greater performance.' button. Below it, under 'Essentials', the 'Url' is listed as <https://contoso-search-svc.search.windows.net>. Other details shown include Status (Running), Location (West US 2), Subscription, and Tags. A red box highlights the 'Url' field.

2. Look up the IP address of the search service by performing a `nslookup` (or a `ping`) of the FQDN on a command prompt. Make sure you remove the `https://` prefix from the FQDN.
3. Copy the IP address so that you can specify it on an inbound rule in the next step. In the following example, the IP address that you should copy is "150.0.0.1".

```
Bash

nslookup contoso.search.windows.net
Server: server.example.org
Address: 10.50.10.50

Non-authoritative answer:
Name: <name>
Address: 150.0.0.1
aliases: contoso.search.windows.net
```

Allow access from your client IP address

Client applications that push indexing and query requests to the search service must be represented in an IP range. On Azure, you can generally determine the IP address by pinging the FQDN of a service (for example, `ping <your-search-service-name>.search.windows.net` returns the IP address of a search service).

Add your client IP address to allow access to the service from the Azure portal on your current computer. Navigate to the **Networking** section on the left pane. Change **Public Network**

Access to Selected networks, and then check Add your client IP address under Firewall.

The screenshot shows a section of the Azure portal's configuration interface. At the top, there are three tabs: 'Firewalls and virtual networks' (which is selected), 'Private endpoint connections', and 'Shared private access'. Below the tabs, a descriptive text states: 'Public endpoints allow access to this resource through the internet using a public IP address. An application or resource that is granted access with the following network rules still requires proper authorization to access this resource.' Underneath this text, there is a configuration section for 'Public network access'. It includes three options: 'All networks' (radio button is empty), 'Selected IP addresses' (radio button is checked and highlighted with a red box), and 'Disabled' (radio button is empty). The 'Selected IP addresses' option is the one being highlighted.

Get the Azure portal IP address

If you're using the Azure portal or the [Import Data wizard](#) to create an indexer, you need an inbound rule for the Azure portal as well.

To get the Azure portal's IP address, perform `nslookup` (or `ping`) on `stamp2.ext.search.windows.net`, which is the domain of the traffic manager. For nslookup, the IP address is visible in the "Non-authoritative answer" portion of the response.

In the following example, the IP address that you should copy is "52.252.175.48".

```
Bash

$ nslookup stamp2.ext.search.windows.net
Server: ZenWiFi_ET8-0410
Address: 192.168.50.1

Non-authoritative answer:
Name: azsyrie.northcentralus.cloudapp.azure.com
Address: 52.252.175.48
Aliases: stamp2.ext.search.windows.net
          azs-ux-prod.trafficmanager.net
          azspncuux.management.search.windows.net
```

Services in different regions connects to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

For ping, the request times out, but the IP address is visible in the response. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is "52.252.175.48".

Get IP addresses for "AzureCognitiveSearch" service tag

You'll also need to create an inbound rule that allows requests from the [multitenant execution environment](#). This environment is managed by Microsoft and it's used to offload processing intensive jobs that could otherwise overwhelm your search service. This section explains how to get the range of IP addresses needed to create this inbound rule.

An IP address range is defined for each region that supports Azure AI Search. Specify the full range to ensure the success of requests originating from the multitenant execution environment.

You can get this IP address range from the `AzureCognitiveSearch` service tag.

1. Use either the [discovery API](#) or the [downloadable JSON file](#). If the search service is in the Azure Public cloud, download the [Azure Public JSON file](#).
2. Open the JSON file and search for "AzureCognitiveSearch". For a search service in WestUS2, the IP addresses for the multitenant indexer execution environment are:

```
JSON

{
  "name": "AzureCognitiveSearch.WestUS2",
  "id": "AzureCognitiveSearch.WestUS2",
  "properties": {
    "changeNumber": 1,
    "region": "westus2",
    "regionId": 38,
    "platform": "Azure",
    "systemService": "AzureCognitiveSearch",
    "addressPrefixes": [
      "20.42.129.192/26",
      "40.91.93.84/32",
      "40.91.127.116/32",
      "40.91.127.241/32",
      "51.143.104.54/32",
      "51.143.104.90/32",
      "2603:1030:c06:1::180/121"
    ],
    "networkFeatures": null
  }
},
```

3. For IP addresses have the "/32" suffix, drop the "/32" (40.91.93.84/32 becomes 40.91.93.84 in the rule definition). All other IP addresses can be used verbatim.

4. Copy all of the IP addresses for the region.

Add IP addresses to IP firewall rules

Now that you have the necessary IP addresses, you can set up the inbound rules. The easiest way to add IP address ranges to a storage account's firewall rule is through the Azure portal.

1. Locate the storage account on the Azure portal and open **Networking** on the left pane.
2. In the **Firewall and virtual networks** tab, choose **Selected networks**.

The screenshot shows the Azure portal interface for a storage account named 'contoso-storage'. The left sidebar has 'Networking' highlighted. The main content area is titled 'contoso-storage | Networking'. It shows the 'Firewalls and virtual networks' tab is selected. Under 'Allow access from', the 'Selected networks' radio button is selected, indicated by a red box. A note below says 'All networks, including the internet, can access this storage account.' At the bottom, there are sections for 'Network Routing' and 'Publish route-specific endpoints'.

3. Add the IP addresses obtained previously in the address range and select **Save**. You should have rules for the search service, Azure portal (optional), plus all of the IP addresses for the "AzureCognitiveSearch" service tag for your region.

The screenshot shows the Azure portal interface for a storage account named 'contoso-storage'. The left sidebar has 'Networking' highlighted. The main content area is titled 'contoso-storage | Networking'. It shows the 'Firewall' section where an IP address range is being added. A red box highlights the 'Address range' input field containing '<YOUR IP ADDRESS HERE>'. Below it is another input field for 'IP address or CIDR'.

It can take five to ten minutes for the firewall rules to be updated, after which indexers should be able to access storage account data behind the firewall.

Supplement network security with token authentication

Firewalls and network security are a first step in preventing unauthorized access to data and operations. Authorization should be your next step.

We recommend role-based access, where Microsoft Entra ID users and groups are assigned to roles that determine read and write access to your service. See [Connect to Azure AI Search using role-based access controls](#) for a description of built-in roles and instructions for creating custom roles.

If you don't need key-based authentication, we recommend that you disable API keys and use role assignments exclusively.

Next Steps

- [Configure Azure Storage firewalls](#)
- [Configure an IP firewall for Azure Cosmos DB](#)
- [Configure IP firewall for Azure SQL Server](#)

Make indexer connections to Azure Storage as a trusted service

In Azure AI Search, indexers that access Azure blobs can use the [trusted service exception](#) to securely access blobs. This mechanism offers customers who are unable to grant [indexer access using IP firewall rules](#) a simple, secure, and free alternative for accessing data in storage accounts.

(!) Note

If Azure Storage is behind a firewall and in the same region as Azure AI Search, you won't be able to create an inbound rule that admits requests from your search service. The solution for this scenario is for search to connect as a trusted service, as described in this article.

Prerequisites

- A search service with a system-assigned managed identity. See [Check service identity](#).
- A storage account with the **Allow trusted Microsoft services to access this storage account** network option. See [Check network settings](#).
- An Azure role assignment in Azure Storage that grants permissions to the search service system-assigned managed identity. See [Check permissions](#).

(!) Note

In Azure AI Search, a trusted service connection is limited to blobs and ADLS Gen2 on Azure Storage. It's unsupported for indexer connections to Azure Table Storage and Azure Files.

A trusted service connection must use a system-assigned managed identity. A user-assigned managed identity isn't currently supported for this scenario.

Check service identity

1. Sign in to the [Azure portal](#)  and [find your search service](#) .
2. From the left pane, select **Settings > Identity**.

3. **Enable a system-assigned identity.** Remember that user-assigned managed identities don't work for a trusted service connection.

my-search-service | Identity

System assigned User assigned

A system assigned managed identity is restricted to one per resource and is tied to the lifecycle of this resource. You can grant permissions to the managed identity by using Azure role-based access control (Azure RBAC). The managed identity is authenticated with Microsoft Entra ID, so you don't have to store any credentials in code.

Status: On

Object (principal) ID: [REDACTED]

Permissions: Azure role assignments

This resource is registered with Microsoft Entra ID. The managed identity can be configured to allow access to other resources. Be careful when making changes to the access settings for the managed identity because it can result in failures. [Learn more](#)

Identity

Check network settings

1. Sign in to the [Azure portal](#) and find your storage account.
2. From the left pane, select **Security + networking > Networking**.
3. On the **Public access** tab, select **Manage**.

my-storage-account | Networking

Public access Private endpoints Network routing Custom domain

Configure access to this storage account using Virtual Networks, IP address ranges, or a network security perimeter. [Learn more](#)

Associate a network security perimeter to secure public network access. [View recommendations](#)

Public network access: Enabled from all networks

Manage

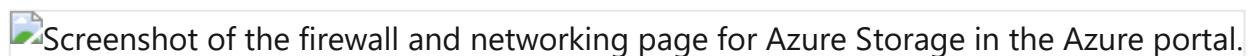
Network security perimeter
Associate a network security perimeter to centrally manage inbound and outbound access rules. [Learn more](#)
No network security perimeter has been associated
Associate

Resource settings: Virtual networks, IP addresses, and exceptions
Configure access rules to specify which networks can access this storage account. [Learn more](#)
Resource settings are not in effect. Public network access is enabled from all networks.

4. Under Public network access scope, select Enable from selected networks.

The screenshot shows the 'Public network access' configuration page. It includes sections for enabling inbound and outbound access, selecting a perimeter, and a note about public access availability. The 'Public network access scope' section is highlighted with a red border around the 'Enable from selected networks' option, indicating it is the correct choice.

5. Under Exceptions, select Allow trusted Microsoft services to access this resource.



Assuming your search service has role-based access to the storage account, it can access data even when connections to Azure Storage are secured by IP firewall rules.

Check permissions

A system-assigned managed identity is a Microsoft Entra service principal. The assignment needs **Storage Blob Data Reader** at a minimum.

1. In the left pane under **Access Control**, view all role assignments and make sure that **Storage Blob Data Reader** is assigned to the search service system identity.
2. Add **Storage Blob Data Contributor** if write access is required.

Features that require write access include [enrichment caching](#), [debug sessions](#), and [knowledge store](#).

Set up and test the connection

The easiest way to test the connection is by running the **Import data** wizard.

1. Start the **Import data** wizard, selecting Azure Blob Storage or Azure Data Lake Storage Gen2.
2. Choose a connection to your storage account, and then select **System-assigned**. Select **Next** to invoke a connection. If the index schema is detected, the connection succeeded.



Related content

- [Connect to other Azure resources using a managed identity](#)
 - [Azure blob indexer](#)
 - [ADLS Gen2 indexer](#)
 - [Authenticate with Microsoft Entra ID](#)
 - [About managed identities \(Microsoft Entra ID\)](#)
-

Last updated on 12/01/2025

Make outbound connections through a shared private link

09/26/2025

This article explains how to configure private, outbound calls from Azure AI Search to an Azure resource that runs within an Azure virtual network.

Setting up a private connection allows a search service to connect to a virtual network IP address instead of a port that's open to the internet. The object created for the connection is called a *shared private link*. On the connection, the search service uses the shared private link internally to reach an Azure resource inside the network boundary.

Shared private link is a premium feature that's billed by usage. When you set up a shared private link, charges for the private endpoint are added to your Azure invoice. As you use the shared private link, data transfer rates for inbound and outbound access are also invoiced. For details, see [Azure Private Link pricing](#).

(!) Note

If you're setting up a private indexer connection to a SQL Managed Instance, see [this article](#) instead for steps specific to that resource type.

When to use a shared private link

Azure AI Search makes outbound calls to other Azure resources in the following scenarios:

- Knowledge agent connections to Azure OpenAI for agentic retrieval workflows
- Indexer or query connections to Azure OpenAI or Azure AI Vision for vectorization
- Indexer connections to supported data sources
- Indexer (skillset) connections to Azure Storage for caching enrichments, debug session state, or writing to a knowledge store
- Indexer (skillset) connections to Azure AI services for billing purposes
- Encryption key requests to Azure Key Vault
- Custom skill requests to Azure Functions or similar resource

Shared private links only work for Azure-to-Azure connections. If you're connecting to OpenAI or another external model provider, the connection must be over the public internet.

Shared private links are for operations and data accessed through a [private endpoint](#) for Azure resources or clients that run in an Azure virtual network.

A shared private link is:

- Created using Azure AI Search tooling, APIs, or SDKs
- Approved by the Azure resource owner
- Used internally by Azure AI Search on a private connection to a specific Azure resource

Only your search service can use the private links that it creates, and there can be only one shared private link created on your service for each resource and subresource combination.

Once you set up the private link, it's used automatically whenever the search service connects to that resource. You don't need to modify the connection string or alter the client you're using to issue the requests, although the device used for the connection must connect using an authorized IP in the Azure resource's firewall.

There are two scenarios for using [Azure Private Link](#) and Azure AI Search together.

- Scenario one: create a shared private link when an *outbound* (indexer or knowledge agent) connection to Azure requires a private connection.
- Scenario two: [configure search for a private *inbound* connection](#) from clients that run in a virtual network.

Scenario one is covered in this article.

While both scenarios have a dependency on Azure Private Link, they're independent. You can create a shared private link without having to configure your own search service for a private endpoint.

Limitations

When evaluating shared private links for your scenario, remember these constraints.

- Several of the resource types used in a shared private link are in preview. If you're connecting to a preview resource (Azure Database for MySQL or Azure SQL Managed Instance), use a preview version of the Management REST API to create the shared private link. These versions include `2020-08-01-preview`, `2021-04-01-preview`, `2024-03-01-preview`, `2024-06-01-preview`, and `2025-02-01-preview`. We recommend the latest preview API.
- Indexer execution must use the [private execution environment](#) that's specific to your search service. Private endpoint connections aren't supported from the multitenant content processing environment. The configuration setting for this requirement is covered in this article.

- Review shared private link **resource limits** for each tier.

Prerequisites

- A [supported Azure resource](#), configured to run in a virtual network.
- An Azure AI Search service with tier and region requirements, by workload:

[\[+\] Expand table](#)

Workload	Tier requirements	Region requirements	Service creation requirements
Indexers without skillsets	Basic and higher	None	None
Skillsets with embedding skills (integrated vectorization)	Basic and higher	High capacity regions	After April 3, 2024
Skillsets using other built-in or custom skills	Standard 1 (S1) and higher	None	After April 3, 2024
Knowledge agents calling Azure OpenAI for query planning or passing search results	Basic and higher	None	None

- Permissions on both Azure AI Search and the Azure resource:

[\[+\] Expand table](#)

Resource	Permissions
Azure AI Search	<code>Microsoft.Search/searchServices/sharedPrivateLinkResources/write</code> <code>Microsoft.Search/searchServices/sharedPrivateLinkResources/read</code> <code>Microsoft.Search/searchServices/sharedPrivateLinkResources/operationStatuses/read</code>
Other Azure resource	Permission to approve private endpoint connections. For example, on Azure Storage, you need <code>Microsoft.Storage/storageAccounts/privateEndpointConnectionsApproval/action</code> .

Supported resource types

You can create a shared private link for the following resources.

[\[+\] Expand table](#)

Resource type	Subresource (or Group ID)
Microsoft.Storage/storageAccounts ¹	blob, table, dfs, file
Microsoft.DocumentDB/databaseAccounts ²	Sql
Microsoft.Sql/servers ³	sqlServer
Microsoft.KeyVault/vaults	vault
Microsoft.DBforMySQL/servers (preview)	mysqlServer
Microsoft.Web/sites ⁴	sites
Microsoft.Sql/managedInstances (preview) ⁵	managedInstance
Microsoft.CognitiveServices/accounts ^{6 7}	openai_account
Microsoft.CognitiveServices/accounts ⁸	cognitiveservices_account
Microsoft.Fabric/privateLinkServicesForFabric ⁹	workspace

¹ If Azure Storage and Azure AI Search are in the same region, the connection to storage is made over the Microsoft backbone network, which means a shared private link is redundant for this configuration. However, if you already set up a private endpoint for Azure Storage, you should also set up a shared private link or the connection is refused on the storage side. Also, if you're using multiple storage formats for various scenarios in search, make sure to create a separate shared private link for each subresource.

² The `Microsoft.DocumentDB/databaseAccounts` resource type is used for indexer connections to Azure Cosmos DB for NoSQL. The provider name and group ID are case-sensitive.

³ The `Microsoft.Sql/servers` resource type is used for connections to Azure SQL database. There's currently no support for a shared private link to Azure Synapse SQL.

⁴ The `Microsoft.Web/sites` resource type is used for App service and Azure functions. In the context of Azure AI Search, an Azure function is the more likely scenario. An Azure function is commonly used for hosting the logic of a custom skill. Azure Function has Consumption, Premium, and Dedicated [App Service hosting plans](#). The [App Service Environment \(ASE\)](#), [Azure Kubernetes Service \(AKS\)](#) and [Azure API Management](#) aren't supported at this time.

⁵ See [Create a shared private link for a SQL Managed Instance](#) for instructions.

⁶ The `Microsoft.CognitiveServices/accounts` resource type is used for vectorizer and indexer connections to Azure OpenAI embedding models when implementing [integrated Vectorization](#). As of November 19, 2024, there's now support for shared private link to support the Azure AI Vision multimodal embeddings via [AI Services multi-service account](#).

⁷ Shared private link for Azure OpenAI is only supported in public cloud and [Microsoft Azure Government](#). Other cloud offerings don't have support for shared private links for `openai_account` Group ID.

⁸ Shared private links are now supported (as of November 2024) for connections to Azure AI services multi-service accounts. Azure AI Search connects to Azure AI services multi-service for [billing purposes](#). These connections can now be private through a shared private link. Shared private link is only supported when configuring [a managed identity \(keyless configuration\)](#) in the skillset definition.

⁹ Shared private link is supported for connections to OneLake workspace. To create a `privateLinkServicesForFabric` resource specific to a workspace, [register Microsoft.Fabric](#) namespace to your subscription and refer to step 2 as documented in [Create the private link service in Azure](#). Note that when using a shared private link, the OneLake data source configuration must be defined with a specific connection string as outlined in the [OneLake indexer documentation](#).

1 - Create a shared private link

Use the Azure portal, Management REST API, the Azure CLI, or Azure PowerShell to create a shared private link.

Here are a few tips:

- Give the private link a meaningful name. In the Azure PaaS resource, a shared private link appears alongside other private endpoints. A name like "shared-private-link-for-search" can remind you how it's used.

When you complete the steps in this section, you have a shared private link that's provisioned in a pending state. **It takes several minutes to create the link.** Once it's created, the resource owner must approve the request before it's operational.

Azure portal

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. Under **Settings** on the left pane, select **Networking**.
3. On the Shared Private Access page, select **+ Add Shared Private Access**.
4. Select either **Connect to an Azure resource in my directory** or **Connect to an Azure resource by resource ID**.

5. If you select the first option (recommended), the Azure portal helps you pick the appropriate Azure resource and fills in other properties, such as the group ID of the resource and the resource type.

The screenshot shows the Azure portal interface for creating a new shared private access link. The left sidebar is for the 'contoso | Networking' service, with 'Networking' selected. The main area shows a table of existing shared private access entries. A modal dialog is open on the right titled 'New Shared Private Access'. In the 'Connection method' section, the radio button for 'Connect to an Azure resource in my directory' is selected. The 'Name' field contains 'blob-pe'. The 'Subscription' dropdown is set to 'Contoso Subscription'. Under 'Resource Type', 'Microsoft.Storage/storageAccounts' is chosen, and the specific resource 'contoso-storage' is selected. The 'Target sub-resource' is set to 'blob'. A 'Request Message' field contains the text 'Please approve'. At the bottom of the dialog are 'Cancel' and 'Create' buttons.

6. If you select the second option, enter the Azure resource ID manually and choose the appropriate group ID from the list at the beginning of this article.

This screenshot shows the same 'New Shared Private Access' dialog as the previous one, but with different configuration. The 'Connection method' is now set to 'Connect to an Azure resource by resource ID or alias'. The 'Name' field remains 'blob-pe'. In the 'Private Link Resource Id' field, the full Azure resource ID is entered: '/subscriptions/00000000-0000-0000-0000-00000000...'. The 'Target sub-resource' is set to 'blob', and the 'Request Message' is 'Please approve'. The 'Create' button is at the bottom.

7. Confirm the provisioning status is "Updating".

Access Control Shared Private Access

Azure Cognitive Search supports private access to other resources using Private Links. Click "Add Shared Private Access" to request a private endpoint for accessing protected resources. Your request must be approved by the resource owner before you can connect. [Learn more](#)

+ Add Shared Private Access Refresh Delete

Name	Resource	Sub-Resource	Provisioning state	Connection state	Message
vault-pe	contoso-vault	vault	Succeeded	Pending	please approve
cosmos-pe	contoso-sql	Sql	Succeeded	Pending	please approve
test-pe	contoso-table	blob	Succeeded	Pending	Please approve
blob-pe	contoso-storage	blob	Updating	Pending	Please approve

- Once the resource is successfully created, the provisioning state of the resource changes to "Succeeded".

Notifications

More events in the activity log → Dismiss all ▾

✓ Azure Cognitive Search notification
Successfully created Shared Private Access: "blob-pe"
2 minutes ago

Shared private link creation workflow

A `202 Accepted` response is returned on success. The process of creating an outbound private endpoint is a long-running (asynchronous) operation. It involves deploying the following resources:

- A private endpoint, allocated with a private IP address in a "Pending" state. The private IP address is obtained from the address space that's allocated to the virtual network of the execution environment for the search service-specific private indexer. Upon approval of

the private endpoint, any communication from Azure AI Search to the Azure resource originates from the private IP address and a secure private link channel.

- A private DNS zone for the type of resource, based on the group ID. By deploying this resource, you ensure that any DNS lookup to the private resource utilizes the IP address that's associated with the private endpoint.

2 - Approve the private endpoint connection

Approval of the private endpoint connection is granted on the Azure PaaS side. Explicit approval by the resource owner is required. The following steps cover approval using the Azure portal, but here are some links to approve the connection programmatically from the Azure PaaS side:

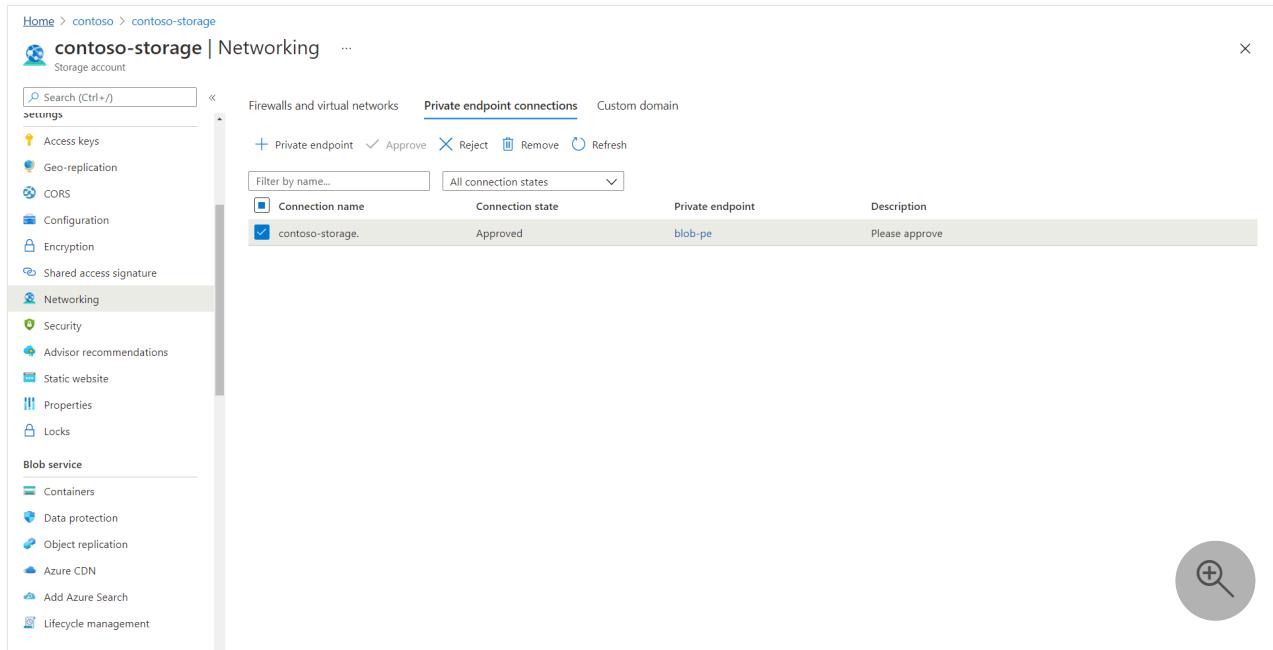
- On Azure Storage, use [Private Endpoint Connections - Put](#)
- On Azure Cosmos DB, use [Private Endpoint Connections - Create Or Update](#)
- On Azure OpenAI, use [Private Endpoint Connections - Create Or Update](#)
- On Microsoft Onelake, use [Private Endpoints - Approve via CLI](#) or [Private Endpoints - Approve via Portal](#)

Using the Azure portal, perform the following steps:

1. Open the **Networking** page of the Azure PaaS resource.[text ↗](#)
2. Find the section that lists the private endpoint connections. The following example is for a storage account.

The screenshot shows the Azure portal interface for a storage account named 'contoso-storage'. The left sidebar has 'Networking' selected under the 'Storage account' category. The main content area is titled 'contoso-storage | Networking'. It displays a table of private endpoint connections. There is one entry: 'contoso-storage.' with a 'Pending' status, a 'blob-pe' private endpoint, and a description 'Please approve'. At the top of the table, there are buttons for '+ Private endpoint', 'Approve', 'Reject', 'Remove', and 'Refresh'. The URL in the browser bar is https://portal.azure.com/#@microsoft.onmicrosoft.com/resource/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.Storage/storageAccounts/contoso-storage/networking.

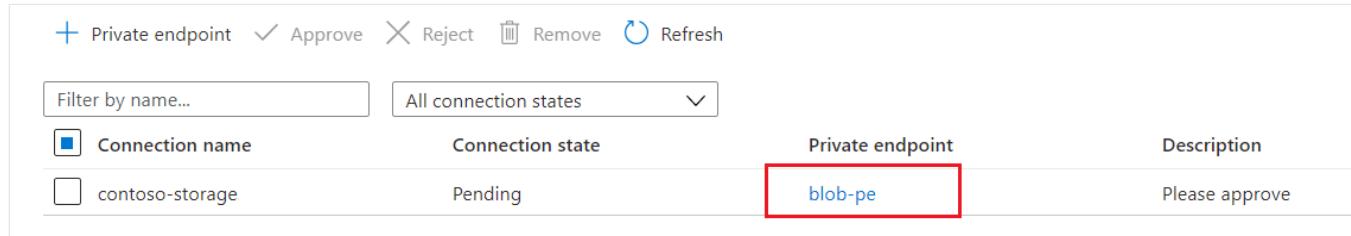
3. Select the connection, and then select **Approve**. It can take a few minutes for the status to be updated in the Azure portal.



The screenshot shows the Azure Storage account 'contoso-storage' Networking page. The left sidebar includes options like Access keys, Geo-replication, CORS, Configuration, Encryption, Shared access signature, and Networking (which is selected). The main area shows a table for 'Private endpoint connections'. A row for 'contoso-storage.' is listed with 'Connection name' checked, 'Connection state' as 'Approved', 'Private endpoint' as 'blob-pe', and 'Description' as 'Please approve'. Action buttons at the top include '+ Private endpoint', 'Approve', 'Reject', 'Remove', and 'Refresh'.

After the private endpoint is approved, Azure AI Search creates the necessary DNS zone mappings in the DNS zone that's created for it.

Although the private endpoint link on the **Networking** page is active, it won't resolve.



The screenshot shows the same Networking page as before, but the connection state for 'contoso-storage.' is now 'Pending'. The 'blob-pe' private endpoint link is highlighted with a red box. The table columns are 'Connection name', 'Connection state', 'Private endpoint', and 'Description'.

Selecting the link produces an error. A status message of "The access token is from the wrong issuer" and "must match the tenant associated with this subscription" appears because the backend private endpoint resource is provisioned by Microsoft in a Microsoft-managed tenant, while the linked resource (Azure AI Search) is in your tenant. It's by design you can't access the private endpoint resource by selecting the private endpoint connection link.

Follow the instructions in the next section to check the status of your shared private link.

3 - Check shared private link status

On the Azure AI Search side, you can confirm request approval by revisiting the Shared Private Access page of the search service **Networking** page. Connection state should be approved.

Search resources, services, and docs (G+)

Home > contoso

contoso | Networking

Access Control Shared Private Access

Azure Cognitive Search supports private access to other resources using Private Links. Click "Add Shared Private Access" to request a private endpoint for accessing protected resources. Your request must be approved by the resource owner before you can connect. [Learn more](#)

+ Add Shared Private Access ⏪ Refresh ⏷ Delete

Filter by name...

Name	Resource	Sub-Resource	Provisioning state	Connection state	Message
vault-pe	contoso-vault	vault	Succeeded	Pending	please approve
cosmos-pe	contoso-sql	Sql	Succeeded	Pending	please approve
test-pe	contoso-table	blob	Succeeded	Pending	Please approve
blob-pe	contoso-storage	blob	Succeeded	Approved	Please approve

Alternatively, you can also obtain connection state by using the [Shared Private Link Resources - Get](#).

.NET CLI

```
az rest --method get --uri https://management.azure.com/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeeee4e4e/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-search/sharedPrivateLinkResources/blob-pe?api-version=2025-09-01
```

This would return a JSON, where the connection state shows up as "status" under the "properties" section. Following is an example for a storage account.

JSON

```
{
  "name": "blob-pe",
  "properties": {
    "privateLinkResourceId": "/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeeee4e4e/resourceGroups/contoso/providers/Microsoft.Storage/storageAccounts/contoso-storage",
    "groupId": "blob",
    "requestMessage": "please approve",
    "status": "Approved",
    "resourceRegion": null,
    "provisioningState": "Succeeded"
  }
}
```

If the provisioning state (`properties.provisioningState`) of the resource is "Succeeded" and connection state(`properties.status`) is "Approved", it means that the shared private link resource is functional and the indexer can be configured to communicate over the private endpoint.

4 - Configure the indexer to run in the private environment

[Indexer execution](#) occurs in either a private environment that's specific to the search service, or a multitenant environment that's used internally to offload expensive skillset processing for multiple customers.

The execution environment is transparent, but once you start building firewall rules or establishing private connections, you must take indexer execution into account. For a private connection, configure indexer execution to always run in the private environment.

This step shows you how to configure the indexer to run in the private environment using the REST API. You can also set the execution environment using the JSON editor in the Azure portal.

Note

You can perform this step before the private endpoint connection is approved. However, until the private endpoint connection shows as approved, any existing indexer that tries to communicate with a secure resource (such as the storage account) will end up in a transient failure state and new indexers will fail to be created.

1. Create the data source definition, index, and skillset (if you're using one) as you would normally. There are no properties in any of these definitions that vary when using a shared private endpoint.
2. [Create an indexer](#) that points to the data source, index, and skillset that you created in the preceding step. In addition, force the indexer to run in the private execution environment by setting the indexer `executionEnvironment` configuration property to `private`.

JSON

```
{  
  "name": "indexer",  
  "dataSourceName": "blob-datasource",  
  "targetIndexName": "index",  
  "parameters": {
```

```
        "configuration": {
            "executionEnvironment": "private"
        }
    },
    "fieldMappings": []
}
```

After the indexer is created successfully, it should connect to the Azure resource over the private endpoint connection. You can monitor the status of the indexer by using the [Indexer Status API](#).

 **Note**

If you already have existing indexers, you can update them via the [PUT API](#) by setting the `executionEnvironment` to `private` or using the JSON editor in the Azure portal.

5 - Test the shared private link

1. If you haven't done so already, verify that your Azure PaaS resource refuses connections from the public internet. If connections are accepted, review the DNS settings in the [Networking](#) page of your Azure PaaS resource.
2. Choose a tool that can invoke an outbound request scenario, such as an indexer connection to a private endpoint. An easy choice is using an [import wizard](#), but you can also try a REST client and REST APIs for more precision. Assuming that your search service isn't also configured for a private connection, the REST client connection to search can be over the public internet.
3. Set the connection string to the private Azure PaaS resource. The format of the connection string doesn't change for shared private link. The search service invokes the shared private link internally.

For indexer workloads, the connection string is in the data source definition. An example of a data source might look like this:

HTTP

```
{
    "name": "my-blob-ds",
    "type": "azureblob",
    "subtype": null,
    "credentials": {
        "connectionString": "DefaultEndpointsProtocol=https;AccountName=<YOUR-
```

```
STORAGE-ACCOUNT>;AccountKey=..."  
    }
```

4. For indexer workloads, remember to set the execution environment in the indexer definition. An example of an indexer definition might look like this:

HTTP

```
"name": "indexer",  
"dataSourceName": "my-blob-ds",  
"targetIndexName": "my-index",  
"parameters": {  
    "configuration": {  
        "executionEnvironment": "private"  
    }  
},  
"fieldMappings": []  
}
```

5. Run the indexer. If the indexer execution succeeds and the search index is populated, the shared private link is working.

Troubleshooting

- If your indexer creation fails with "Data source credentials are invalid," check the approval status of the shared private link before debugging the connection. If the status is `Approved`, check the `properties.provisioningState` property. If it's `Incomplete`, there might be a problem with underlying dependencies. In this case, reissue the `PUT` request to re-create the shared private link. You might also need to repeat the approval step.
- If indexers fail consistently or intermittently, check the `executionEnvironment` property on the indexer. The value should be set to `private`. If you didn't set this property, and indexer runs succeeded in the past, it's because the search service used a private environment of its own accord. A search service moves processing out of the multitenant environment if the system is under load.
- If you get an error when creating a shared private link, check `service limits` to verify that you're under the quota for your tier.

Next steps

Learn more about private endpoints and other secure connection methods:

- [Troubleshoot issues with shared private link resources](#)

- What are private endpoints?
- DNS configurations needed for private endpoints
- Indexer access to content protected by Azure network security features

Create a shared private link for a SQL managed instance from Azure AI Search

Article • 01/28/2025

This article explains how to configure an indexer in Azure AI Search for a private connection to a SQL managed instance that runs within a virtual network. The private connection is through a [shared private link](#) and Azure Private Link.

On a private connection to a managed instance, the fully qualified domain name (FQDN) of the instance must include the [DNS Zone](#). Currently, only the Azure AI Search Management REST API provides a `dnsZonePrefix` parameter for accepting the DNS zone specification.

Although you can call the Management REST API directly, it's easier to use the Azure CLI `az rest` module to send Management REST API calls from a command line. This article uses the Azure CLI with REST to set up the private link.

ⓘ Note

This article refers to Azure portal for obtaining properties and confirming steps.

However, when creating the shared private link for SQL Managed Instance, make

sure you're using the REST API. Although the Networking tab lists

`Microsoft.Sql/managedInstances` as an option, the Azure portal doesn't currently

support the extended URL format used by SQL Managed Instance.

Prerequisites

- [Azure CLI](#)
- Azure AI Search, Basic or higher. If you're using [AI enrichment](#) and skillsets, use Standard 2 (S2) or higher. See [Service limits](#) for details.
- Azure SQL Managed Instance, configured to run in a virtual network.
- You should have a minimum of Contributor permissions on both Azure AI Search and SQL Managed Instance.
- Azure SQL Managed Instance connection string. Managed identity isn't currently supported with shared private link. Your connection string must include a user name and password.

ⓘ Note

Shared private links are billable through [Azure Private Link pricing](#) and charges are invoiced based on usage.

1 - Retrieve connection information

In this section, get the DNS zone from the host name and a connection string.

1. In Azure portal, find the SQL managed instance object.
2. On the **Overview** tab, locate the Host property. Copy the *DNS zone* portion of the FQDN for the next step. The DNS zone is part of the domain name of the SQL Managed Instance. For example, if the FQDN of the SQL Managed Instance is `my-sql-managed-instance.a1b22c333d44.database.windows.net`, the DNS zone is `a1b22c333d44`.
3. On the **Connection strings** tab, copy the ADO.NET connection string for a later step. It's needed for the data source connection when testing the private connection.

For more information about connection properties, see [Create an Azure SQL Managed Instance](#).

2 - Create the body of the request

1. Using a text editor, create the JSON for the shared private link.

JSON

```
{  
  "name": "{{shared-private-link-name}}",  
  "properties": {  
    "privateLinkResourceId": "/subscriptions/{{target-resource-subscription-ID}}/resourceGroups/{{target-resource-rg}}/providers/Microsoft.Sql/managedInstances/{{target-resource-name}}",  
    "dnsZonePrefix": "a1b22c333d44",  
    "groupId": "managedInstance",  
    "requestMessage": "please approve"  
  }  
}
```

Provide a meaningful name for the shared private link. The shared private link appears alongside other private endpoints. A name like "shared-private-link-for-search" can remind you how it's used.

Paste in the DNS zone name in "dnsZonePrefix" that you retrieved in an earlier step.

Edit the "privateLinkResourceId", substitute valid for values for the placeholders. Provide a valid subscription ID, resource group name, and managed instance name.

2. Save the file locally as *create-pe.json* (or use another name, remembering to update the Azure CLI syntax in the next step).
3. In the Azure CLI, type `dir` to note the current location of the file.

3 - Create a shared private link

1. From the command line, sign into Azure using `az login`.
2. If you have multiple subscriptions, make sure you're using the one you intend to use: `az account show`.
To set the subscription, use `az account set --subscription {{subscription ID}}`
3. Call the `az rest` command to use the [Management REST API](#) of Azure AI Search.

Because shared private link support for SQL managed instances is still in preview, you need a preview version of the management REST API. Use `2021-04-01-preview` or a later preview API version for this step. We recommend using the latest preview API version.

Azure CLI

```
az rest --method put --uri
https://management.azure.com/subscriptions/{{search-service-
subscription-ID}}/resourceGroups/{{search service-resource-
group}}/providers/Microsoft.Search/searchServices/{{search-service-
name}}/sharedPrivateLinkResources/{{shared-private-link-name}}?api-
version=2024-06-01-preview --body @create-pe.json
```

Provide the subscription ID, resource group name, and service name of your Azure AI Search resource.

Provide the same shared private link name that you specified in the JSON body.

Provide a path to the `create-pe.json` file if you've navigated away from the file location. You can type `dir` at the command line to confirm the file is in the current directory.

4. Run the command.

When you complete these steps, you should have a shared private link that's provisioned in a pending state. **It takes several minutes to create the link.** Once it's created, the resource owner needs to approve the request before it's operational.

You can check the status of the shared private link in the Azure portal. On your search service page, under **Settings > Properties**, scroll down to find the shared private link resources and view the JSON value. When the provisioning state changes from *pending* to *succeeded*, you can continue on to the next step.

4 - Approve the private endpoint connection

On the SQL Managed Instance side, the resource owner must approve the private connection request you created.

1. In the Azure portal, open the **Security > Private endpoint connections** of the managed instance.
2. Find the section that lists the private endpoint connections.
3. Select the connection, and then select **Approve**. It can take a few minutes for the status to be updated in the Azure portal.

After the private endpoint is approved, Azure AI Search creates the necessary DNS zone mappings in the DNS zone that's created for it.

5 - Check shared private link status

On the Azure AI Search side, you can confirm request approval by revisiting the Shared Private Access tab of the search service **Networking** page. Connection state should be approved.

The screenshot shows the Azure portal interface for a search service named 'contoso'. On the left, there's a navigation sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (which is currently selected), Quick start, Keys, Scale, Search traffic analytics, Identity, Networking (which is also selected), Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic settings, Logs, and Automation. The main content area is titled 'Shared Private Access' under 'Access Control'. It displays a table with four rows of data. The columns are Name, Resource, Sub-Resource, Provisioning state, Connection state, and Message. The 'blob-pe' row is highlighted with a red box around its entire row.

Name	Resource	Sub-Resource	Provisioning state	Connection state	Message
vault-pe	contoso-vault	vault	Succeeded	Pending	please approve
cosmos-pe	contoso-sql	Sql	Succeeded	Pending	please approve
test-pe	contoso-table	blob	Succeeded	Pending	Please approve
blob-pe	contoso-storage	blob	Succeeded	Approved	Please approve

6 - Configure the indexer to run in the private environment

You can now configure an indexer and its data source to use an outbound private connection to your managed instance.

This article assumes a [REST client](#) and uses the REST APIs.

1. [Create the data source definition](#) as you would normally for Azure SQL. By default, a managed instance listens on port 3342, but on a virtual network it listens on 1433.

Provide the connection string that you copied earlier with an Initial Catalog set to your database name.

```
HTTP

POST https://myservice.search.windows.net/datasources?api-version=2024-07-01
Content-Type: application/json
api-key: admin-key
{
    "name" : "my-sql-datasource",
    "description" : "A database for testing Azure AI Search indexes.",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" :
"Server=tcp:contoso.a1b22c333d44.database.windows.net,1433;Persist Security Info=False; User ID=<your user name>; Password=<your password>;MultipleActiveResultsSets=False; Encrypt=True;Connection Timeout=30;Initial Catalog=<your database name>"}
```

```
        },
        "container" : {
            "name" : "Name of table or view to index",
            "query" : null (not supported in the Azure SQL indexer)
        },
        "dataChangeDetectionPolicy": null,
        "dataDeletionDetectionPolicy": null,
        "encryptionKey": null
    }
}
```

2. Create the indexer definition, setting the indexer `executionEnvironment` to "private".

Indexer execution occurs in either a private execution environment that's specific to your search service, or a multitenant environment hosted by Microsoft and used to offload expensive skillset processing for multiple customers. **When connecting over a private endpoint, indexer execution must be private.**

HTTP

```
POST https://myservice.search.windows.net/indexers?api-version=2024-07-01
Content-Type: application/json
api-key: admin-key
{
    "name": "indexer",
    "dataSourceName": "my-sql-datasource",
    "targetIndexName": "my-search-index",
    "parameters": {
        "configuration": {
            "executionEnvironment": "private"
        }
    },
    "fieldMappings": []
}
```

3. Run the indexer. If the indexer execution succeeds and the search index is populated, the shared private link is working.

You can monitor the status of the indexer in Azure portal or by using the [Indexer Status API](#).

You can use [Search explorer](#) in Azure portal to check the contents of the index.

7 - Test the shared private link

If you ran the indexer in the previous step and successfully indexed content from your managed instance, then the test was successful. However, if the indexer fails or there's no content in the index, you can modify your objects and repeat testing by choosing any client that can invoke an outbound request from an indexer.

An easy choice is [running an indexer](#) in Azure portal, but you can also try a [REST client](#) and REST APIs for more precision. Assuming that your search service isn't also configured for a private connection, the REST client connection to Azure AI Search can be over the public internet.

Here are some reminders for testing:

- If you use a REST client, use the [Management REST API](#) and the [2021-04-01-Preview API version](#) to create the shared private link. Use the [Search REST API](#) and a [stable API version](#) to create and invoke indexers and data sources.
- You can use the Import data wizard to create an indexer, data source, and index. However, the generated indexer won't have the correct execution environment setting.
- You can edit data source and indexer JSON in Azure portal to change properties, including the execution environment and the connection string.
- You can reset and rerun the indexer in Azure portal. Reset is important for this scenario because it forces a full reprocessing of all documents.
- You can use Search explorer to check the contents of the index.

See also

- [Make outbound connections through a private endpoint](#)
- [Indexer connections to Azure SQL Managed Instance through a public endpoint](#)
- [Index data from Azure SQL](#)
- [Management REST API](#)
- [Search REST API](#)
- [Quickstart: Get started with REST](#)

Feedback

Was this page helpful?

 Yes

 No

Document-level access control in Azure AI Search

Azure AI Search supports document-level access control, enabling organizations to enforce fine-grained permissions at the document level, from data ingestion through query execution. This capability is essential for building secure AI agentic systems grounding data, Retrieval-Augmented Generation (RAG) applications, and enterprise search solutions that require authorization checks at the document level.

Approaches for document-level access control

[] Expand table

Approach	Description
Security filters	<p>String comparison. Your application passes in a user or group identity as a string, which populates a filter on a query, excluding any documents that don't match on the string.</p> <p>Security filters are a technique for achieving document-level access control. This approach isn't bound to an API so you can use any version or package.</p>
POSIX-like ACL / RBAC scopes (preview)	<p>Microsoft Entra security principal behind the query token is compared to the permission metadata of documents returned in search results, excluding any documents that don't match on permissions. Access Control Lists (ACL) permissions apply to Azure Data Lake Storage (ADLS) Gen2 directories and files. Role-based access control (RBAC) scopes apply to ADLS Gen2 content and to Azure blobs.</p> <p>Built-in support for identity-based access at the document level is in preview, available in REST APIs and preview Azure SDK packages that provide the feature. Be sure to check the SDK package change log for evidence of feature support.</p>
Microsoft Purview sensitivity labels (preview)	<p>Indexer extracts sensitivity labels defined in Microsoft Purview from supported data sources (Azure Blob Storage, ADLS Gen2, SharePoint in Microsoft 365, OneLake). These labels are stored as metadata and evaluated at query time to enforce user access based on Microsoft Entra tokens and Purview policy assignments. This approach aligns Azure AI Search authorization with your enterprise's Microsoft Information Protection model.</p>
SharePoint in Microsoft 365 ACLs (preview)	<p>When configured, Azure AI Search indexers extract SharePoint document permissions directly from in Microsoft 365 ACLs during initial ingestion. Access checks use Microsoft Entra user and group memberships. Supported group types include Microsoft Entra security groups, Microsoft 365 groups, and mail-enabled security groups. SharePoint groups are not yet supported in preview.</p>

Pattern for security trimming using filters

For scenarios where native ACL/RBAC scopes integration isn't viable, we recommend security string filters for trimming results based on exclusion criteria. The pattern includes the following components:

- To store user or group identities, create a string field in the index.
- Load the index using source documents that include associated ACLs.
- Include a filter expression in your query logic for matching on the string.
- At query time, get the identity of the caller.
- Pass in the identity of the caller as the filter string.
- Results are trimmed to exclude any matches that fail to include the user or group identity string,

You can use push or pull model APIs. Because this approach is API agnostic, you just need to ensure that the index and query have valid strings (identities) for the filtration step.

This approach is useful for systems with custom access models or non-Microsoft security frameworks. For more information this approach, see [Security filters for trimming results in Azure AI Search](#).

Pattern for native support for POSIX-like ACL and RBAC scope permissions (preview)

Native support is based on Microsoft Entra users and groups affiliated with documents that you want to index and query.

Azure Data Lake Storage (ADLS) Gen2 containers support ACLs on the container and on files. For ADLS Gen2, RBAC scope preservation at document level is natively supported when you use an [ADLS Gen2 indexer](#) or a [Blob knowledge source \(supports ADLS Gen2\)](#) and a preview API to ingest content. For Azure blobs using the [Azure blob indexer](#) or knowledge source, RBAC scope preservation is at the container level.

For ACL-secured content, we recommend group access over individual user access for ease of management. The pattern includes the following components:

- Start with documents or files that have ACL assignments.
- [Enable permission filters](#) in the index.
- [Add a permission filter](#) to a string field in an index.
- Load the index with source documents having associated ACLs.
- Query the index, adding [x-ms-query-source-authorization](#) in the request header.

Your client app receives read permissions to the index through **Search Index Data Reader** or **Search Index Data Contributor** role. Access at query time is determined by user or group permission metadata in the indexed content. Queries that include a permission filter pass a user or group token as `x-ms-query-source-authorization` in the request header. When you use permission filters at query time, Azure AI Search checks for two things:

- First, it checks for **Search Index Data Reader** permission that allows your client application to access the index.
- Second, given the extra token on the request, it checks for user or group permissions on documents that are returned in search results, excluding any that don't match.

To get permission metadata into the index, you can use the push model API, pushing any JSON documents to the search index, where the payload includes a string field providing POSIX-like ACLs for each document. The important difference between this approach and security trimming is that the permission filter metadata in the index and query is recognized as Microsoft Entra ID authentication, whereas the security trimming workaround is simple string comparison. Also, you can use the Graph SDK to retrieve the identities.

You can also use the pull model (indexer) APIs if the data source is [Azure Data Lake Storage \(ADLS\) Gen2](#) and your code calls a preview API for indexing.

Retrieve ACL permissions metadata during data ingestion process (preview)

How you retrieve ACL permissions varies depending on whether you're pushing a documents payload or using the ADLS Gen2 indexer.

Start with a preview API that provides the feature:

- [2025-11-01-preview REST API](#)
- [Azure SDK for Python prerelease package ↗](#)
- [Azure SDK for .NET prerelease package ↗](#)
- [Azure SDK for Java prerelease package ↗](#)

For the [push model approach](#):

1. Ensure your index schema is also created with a preview or prerelease SDK and that the schema has permission filters.
2. Consider using the Microsoft Graph SDK to get group or user identities.
3. Use the [Index Documents](#) or equivalent Azure SDK API to push documents and their associated permission metadata into the search index.

For the [pull model ADLS Gen2 indexer approach](#) or [Blob \(ADLS Gen2\) knowledge source](#):

1. Verify that files in the directory are secured using the [ADLS Gen2 access control model](#).
2. Use the [Create Indexer REST API](#) or [Create Knowledge Source REST API](#) or equivalent Azure SDK API to create the indexer, index, and data source.

Pattern for SharePoint in Microsoft 365 basic ACL permissions ingestion (preview)

For SharePoint in Microsoft 365 content, Azure AI Search can apply document-level permissions based on SharePoint ACLs. This integration promotes that only users or groups with access to the source document in SharePoint can retrieve it in search results, as soon as the permissions are synchronized in the index. Permissions are applied to the index either during initial document ingestion.

SharePoint ACL support is available in preview through the SharePoint indexer using the [2025-11-01-preview REST API](#) or supported SDK. The indexer extracts file and list item permission metadata and preserves it in the search index, where it's used to enforce access control at query time.

The pattern includes the following components:

- Use the SharePoint in Microsoft 365 indexer with application permissions to read SharePoint site content and full permissions to read ACLs. Follow the [SharePoint indexer ACL setup instructions](#) for enablement and limitations.
- During initial indexing, SharePoint ACL entries (users and groups) are stored as permission metadata in the search index.
- For incremental indexing of ACLs, review the [SharePoint ACL resync available mechanisms](#) available during public preview.
- At query time, Azure AI Search checks the Microsoft Entra principal in the query token against SharePoint ACL metadata stored in the index. It excludes any documents the caller isn't authorized to access.

During preview, only the following principal types are supported in SharePoint ACLs:

- Microsoft Entra user accounts
- Microsoft Entra security groups
- Microsoft 365 groups
- Mail-enabled security groups

SharePoint groups aren't supported in the preview release.

For configuration details and full limitations, see [How to index SharePoint in Microsoft 365 document-level permissions \(preview\)](#).

Pattern for Microsoft Purview sensitivity labels (preview)

Azure AI Search can ingest and enforce **Microsoft Purview sensitivity labels** for document-level access control, extending information protection policies from Microsoft Purview into your search and retrieval applications.

When label ingestion is enabled, Azure AI Search extracts sensitivity metadata from supported data sources. These include: Azure Blob Storage, Azure Data Lake Storage Gen2 (ADLS Gen2), SharePoint in Microsoft 365, and Microsoft OneLake. The extracted labels are stored in the index alongside document content.

At query time, Azure AI Search checks each document's sensitivity label, the user's Microsoft Entra token, and the organization's Purview policies to determine access. Documents are returned only if the user's identity and label-based permissions allow access under the configured Purview policies.

The pattern includes the following components:

- Configure your [index](#), [data source](#) and [indexer](#) (for scheduling purposes) using the 2025-11-01-preview REST API or a corresponding SDK that supports Purview label ingestion.
- Enable a [system-assigned managed identity](#) to your search service. Then ask your tenant global administrator or privileged role administrator to [grant the required access](#), so the search service can securely access Microsoft Purview and extract label metadata.
- Apply sensitivity labels to documents before indexing so they can be recognized and preserved during ingestion.
- At query time, attach a valid Microsoft Entra token via the header `x-ms-query-source-authorization` to each query request. Azure AI Search evaluates the token and the associated label metadata to enforce label-based access control.

Purview sensitivity label enforcement is limited to single-tenant scenarios, requires RBAC authentication, and during public preview is supported only through REST API or SDK. Autocomplete and Suggest APIs aren't available for Purview-enabled indexes at this time.

For more information, see [Use Azure AI Search indexers to ingest Microsoft Purview sensitivity labels](#).

Enforce document-level permissions at query time

With native [token-based querying](#), Azure AI Search validates a user's [Microsoft Entra token](#), trimming result sets to include only documents the user is authorized to access.

You can achieve automatic trimming by attaching the user's Microsoft Entra token to your query request. For more information, see [Query-time ACL and RBAC enforcement in Azure AI Search](#).

Benefits of document-level access control

Document-level access control is critical for safeguarding sensitive information in AI-driven applications. It helps organizations build systems that align with their access policies, reducing the risk of exposing unauthorized or confidential data. By integrating access rules directly into the search pipeline, AI systems can provide responses grounded in secure and authorized information.

By offloading permission enforcement to Azure AI Search, developers can focus on building high-quality retrieval and ranking systems. This approach helps reducing the need to handle nested groups, write custom filters, or manually trim search results.

Document-level permissions in Azure AI Search provide a structured framework for enforcing access controls that align with organizational policies. By using Microsoft Entra-based ACLs and RBAC roles, organizations can create systems that support robust compliance and promote trust among users. These built-in capabilities reduce the need for custom coding, offering a standardized approach to document-level security.

Tutorials and samples

Take a closer look at document-level access control in Azure AI Search with more articles and samples.

- [Tutorial: Index ADLS Gen2 permissions metadata using an indexer](#)
- [azure-search-rest-samples/Quickstart-ACL ↗](#)
- [azure-search-python-samples/Quickstart-Document-Permissions-Push-API ↗](#)
- [azure-search-python-samples/Quickstart-Document-Permissions-Pull-API ↗](#)
- [Demo app: Ingesting and honoring sensitivity labels ↗](#)

Related content

- [How to index document-level permissions using push API](#)
- [How to index document-level permissions using the ADLS Gen2 indexer](#)
- [How to index document-level permissions using the SharePoint in Microsoft 365 indexer ↗](#)
- [How to index sensitivity labels using indexers](#)
- [How to query using Microsoft Entra token-based permissions ↗](#)

Last updated on 11/18/2025

Security filters for trimming results in Azure AI Search

07/16/2025

For search solutions that can't use the [built-in access control list \(ACL\) support](#) for document-level authorization, Azure AI Search supports creating a filter that trims search results based on a string containing a group or user identity.

This article describes a pattern for security filtering having the following steps:

- ✓ Assemble source documents with the required content, including a string for storing an identity
- ✓ Create a field in the search index for the principal identifiers
- ✓ Push the documents to the search index for indexing
- ✓ Query the index with the `search.in` filter function

It concludes with links to demos and examples that provide hands-on learning. We recommend reviewing this article first to understand the pattern.

About the security filter pattern

The security filter pattern simulates document-level authorization by using a regular OData filter that includes or excludes a search result based on a string consisting of a security principal. There's no authentication or authorization through the security principal. The principal is just a string, used in a filter expression, to include or exclude a document from the search results.

There are several ways to achieve security filtering. One way is through a complicated disjunction of equality expressions: for example, `Id eq 'id1' or Id eq 'id2'`, and so forth. This approach is error-prone, difficult to maintain, and in cases where the list contains hundreds or thousands of values, slows down query response time by many seconds.

A better solution is using the `search.in` function for security filters, as described in this article. If you use `search.in(Id, 'id1, id2, ...')` instead of an equality expression, you can expect subsecond response times.

Prerequisites

- A string field containing a group or user identity, such as a Microsoft Entra object identifier.

- Other fields in the same document should provide the content that's accessible to that group or user. In the following JSON documents, the "security_id" fields contain identities used in a security filter, and the name, salary, and marital status are included if the identity of the caller matches the "security_id" of the document.

```
JSON

{
    "Employee-1": {
        "employee_id": "100-1000-10-1-10000-1",
        "name": "Abram",
        "salary": 75000,
        "married": true,
        "security_id": "alphanumeric-object-id-for-employee-1"
    },
    "Employee-2": {
        "employee_id": "200-2000-20-2-20000-2",
        "name": "Adams",
        "salary": 75000,
        "married": true,
        "security_id": "alphanumeric-object-id-for-employee-2"
    }
}
```

Create security field

In the search index, within the fields collection, you need one field that contains the group or user identity, similar to the fictitious "security_id" field in the previous example.

1. Add a security field as a `Collection(Edm.String)`.
2. Set the field's `filterable` attribute set to `true`.
3. Set the field's `retrievable` attribute to `false` so that it isn't returned as part of the search request.
4. Indexes require a document key. The "file_id" field satisfies that requirement.
5. Indexes should also contain searchable and retrievable content. The "file_name" and "file_description" fields represent that in this example.

The following index schema satisfies the field requirements. Documents that you index on Azure AI Search should have values for all of these fields, including the "group_ids". For the document with `file_name` "secured_file_b", only users that belong to group IDs "group_id1" or "group_id2" have read access to the file.

```
https
```

```
POST https://[search
service].search.windows.net/indexes/securedfiles/docs/index?api-version=2024-
07-01
{
    "name": "securedfiles",
    "fields": [
        {"name": "file_id", "type": "Edm.String", "key": true, "searchable": false },
        {"name": "file_name", "type": "Edm.String", "searchable": true },
        {"name": "file_description", "type": "Edm.String", "searchable": true },
        {"name": "group_ids", "type": "Collection(Edm.String)" },
        "filterable": true, "retrievable": false }
    ]
}
```

Push data into your index using the REST API

Populate your search index with documents that provide values for each field in the fields collection, including values for the security field. Azure AI Search doesn't provide APIs or features for populating the security field specifically. However, several of the examples listed at the end of this article explain techniques for populating this field.

In Azure AI Search, the approaches for loading data are:

- A single push or pull (indexer) operation that imports documents populated with all fields
- Multiple push or pull operations. As long as secondary import operations target the right document identifier, you can load fields individually through multiple imports.

The following example shows a single HTTP POST request to the docs collection of your index's URL endpoint (see [Documents - Index](#)). The body of the HTTP request is a JSON rendering of the documents to be indexed:

```
HTTP
```

```
POST https://[search service].search.windows.net/indexes/securedfiles/docs/index?
api-version=2024-07-01
{
    "value": [
        {
            "@search.action": "upload",
            "file_id": "1",
            "file_name": "secured_file_a",
            "file_description": "File access is restricted to Human Resources.",
            "group_ids": ["group_id1"]
        },
    ],
}
```

```
{
    "@search.action": "upload",
    "file_id": "2",
    "file_name": "secured_file_b",
    "file_description": "File access is restricted to Human Resources and Recruiting.",
    "group_ids": ["group_id1", "group_id2"]
},
{
    "@search.action": "upload",
    "file_id": "3",
    "file_name": "secured_file_c",
    "file_description": "File access is restricted to Operations and Logistics.",
    "group_ids": ["group_id5", "group_id6"]
}
]
```

If you need to update an existing document with the list of groups, you can use the `merge` or `mergeOrUpload` action:

JSON

```
{
    "value": [
        {
            "@search.action": "mergeOrUpload",
            "file_id": "3",
            "group_ids": ["group_id7", "group_id8", "group_id9"]
        }
    ]
}
```

Apply the security filter in the query

In order to trim documents based on `group_ids` access, you should issue a search query with a `group_ids/any(g:search.in(g, 'group_id1, group_id2,...'))` filter, where 'group_id1, group_id2,...' are the groups to which the search request issuer belongs.

This filter matches all documents for which the `group_ids` field contains one of the given identifiers. For full details on searching documents using Azure AI Search, you can read [Search Documents](#).

This sample shows how to set up query using a POST request.

Issue the HTTP POST request, specifying the filter in the request body:

HTTP

```
POST https://[service name].search.windows.net/indexes/securedfiles/docs/search?  
api-version=2024-07-01  
  
{  
    "filter": "group_ids/any(g:search.in(g, 'group_id1, group_id2'))"  
}
```

You should get the documents back where `group_ids` contains either "group_id1" or "group_id2". In other words, you get the documents to which the request issuer has read access.

JSON

```
{  
    [  
        {  
            "@search.score":1.0,  
            "file_id": "1",  
            "file_name": "secured_file_a",  
        },  
        {  
            "@search.score":1.0,  
            "file_id": "2",  
            "file_name": "secured_file_b"  
        }  
    ]  
}
```

Next steps

This article describes a pattern for filtering results based on user identity and the `search.in()` function. You can use this function to pass in principal identifiers for the requesting user to match against principal identifiers associated with each target document. When a search request is handled, the `search.in` function filters out search results for which none of the user's principals have read access. The principal identifiers can represent things like security groups, roles, or even the user's own identity.

For more examples, demos, and videos:

- [Get started with chat document security in Python](#)
- [Set up optional sign in and document level access control \(modifications to the AzureOpenAIDemo app\)](#)
- [Video: Secure your Intelligent Applications with Microsoft Entra](#)

Indexing document Access Control Lists (ACLs) using the push REST APIs

09/18/2025

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Indexing documents, along with their associated [Access Control Lists \(ACLs\)](#) and container [Role-Based Access Control \(RBAC\) roles](#), into an Azure AI Search index via the [push REST APIs](#) preserves document-level permission on indexed content at query time.

Key features include:

- Flexible control over ingestion pipelines.
- Standardized schema for permissions metadata.
- Support for hierarchical permissions, such as folder-level ACLs.

This article explains how to use the push REST API to index document-level permissions' metadata in Azure AI Search. This process prepares your index to query and enforce end-user permissions on search results.

Prerequisites

- Content with ACL metadata from [Microsoft Entra ID](#) or another POSIX-style ACL system.
- The [latest preview REST API](#) or a preview Azure SDK package providing equivalent features.
- An index schema with a `permissionFilterOption` enabled, plus `permissionFilter` field attributes that store the permissions associated with the document.

Limitations

- An ACL field with permission filter type `userIds` or `groupIds` can hold at most 32 values.

- An index can hold at most five unique values among fields of type `rbacScope` on all documents. There's no limit on the number of documents that share the same value of `rbacScope`.
- A preexisting field can be converted into a `permissionFilter` field type for use with built-in ACLs or RBAC metadata filtering. To enable filtering on an existing index, create new fields or modify an existing field to include a `permissionFilter`.
- Only one field of each `permissionFilter` type (one each of `groupIds`, `usersIds`, and `rbacScope`) can exist in an index.
- Each `permissionFilter` field should have `filterable` set to true.
- This functionality is currently not supported in the Azure portal.

Create an index with permission filter fields

Indexing document ACLs and RBAC metadata with the REST API requires setting up an index schema that enables permission filters and has fields with permission filter assignments.

First, add a `permissionFilterOption` option. Valid values are `enabled` or `disabled`, and you should set it to `enabled`. You can switch it to `disabled` if you want to turn off the permission filter functionality at the index level.

Second, create string fields for your permission metadata and include `permissionFilter`. Recall that you can have one of each permission filter type.

Here's a basic example schema that includes all `permissionFilter` types:

```
JSON

{
  "fields": [
    { "name": "UserIds", "type": "Collection(Edm.String)", "permissionFilter": "userIds", "filterable": true },
    { "name": "GroupIds", "type": "Collection(Edm.String)", "permissionFilter": "groupIds", "filterable": true },
    { "name": "RbacScope", "type": "Edm.String", "permissionFilter": "rbacScope", "filterable": true },
    { "name": "DocumentId", "type": "Edm.String", "key": true }
  ],
  "permissionFilterOption": "enabled"
}
```

REST API indexing example

Once you have an index with permission filter fields, you can populate those values using the indexing push API as with any other document fields. Here's an example using the specified index schema, where each document specifies the upload action, the key field (DocumentId), and permission fields. It should also have content, but that field is omitted in this example for brevity.

```
https
```

```
POST  
https://exampleservice.search.windows.net/indexes('indexdocumentsexample')/docs/search.index?api-version=2025-08-01-preview  
{  
    "value": [  
        {  
            "@search.action": "upload",  
            "DocumentId": "1",  
            "UserIds": ["00aa00aa-bb11-cc22-dd33-44ee44ee44ee", "11bb11bb-cc22-dd33-  
ee44-55ff55ff55ff", "22cc22cc-dd33-ee44-ff55-66aa66aa66aa"],  
            "GroupIds": ["none"]  
            "RbacScope": "/subscriptions/00000000-0000-0000-0000-  
000000000000/resourceGroups/Example-Storage-  
rg/providers/Microsoft.Storage/storageAccounts/azurestorage12345/blobServices/defa  
ult/containers/blob-container-01"  
        },  
        {  
            "@search.action": "merge",  
            "DocumentId": "2",  
            "UserIds": ["all"],  
            "GroupIds": ["33dd33dd-ee44-ff55-aa66-77bb77bb77bb", "44ee44ee-ff55-aa66-  
bb77-88cc88cc88cc"]  
        },  
        {  
            "@search.action": "mergeOrUpload",  
            "DocumentId": "3",  
            "UserIds": ["1cdd8521-38cf-49ab-b483-17ddaa48f68f"],  
            "RbacScope": "/subscriptions/00000000-0000-0000-0000-  
000000000000/resourceGroups/Example-Storage-  
rg/providers/Microsoft.Storage/storageAccounts/azurestorage12345/blobServices/defa  
ult/containers/blob-container-03"  
        }  
    ]  
}
```

ACL access resolution rules

This section explains how document access is determined for a user based on the ACL values assigned to each document. The key rule is that *a user only needs to match one ACL type to*

gain access to the document. For example, if a document has fields for `userIds`, `groupIds`, and `rbacScope`, the user can access the document by matching any one of these ACL fields.

Special ACL values "all" and "none"

ACL fields, such as `userIds` and `groupIds`, typically contain lists of GUIDs (Globally Unique Identifiers) that identify the users and groups with access to the document. Two special string values, "all" and "none", are supported for these ACL field types. These values act as broad filters to control access at the global level as showcased in the following table.

[] Expand table

userIds / groupIds value	Meaning
<code>["all"]</code>	Any user can access the document
<code>["none"]</code>	No user can access the document by matching this ACL type
<code>[] (empty array)</code>	No user can access the document by matching this ACL type

Because a user needs to match only one field type, the special value "all" grants public access regardless of the contents of any other ACL field, as all users are matched and granted permission. In contrast, setting `userIds` to "none" or "empty" means no users are granted access to the document *based on their user ID*. It might be possible that they're still granted access by matching their group ID or by RBAC metadata.

Access control example

This example illustrates how the document access rules are resolved based on the specific document ACL field values. For readability, this scenario uses ACL aliases such as "user1," "group1," instead of GUIDs.

[] Expand table

Document #	userIds	groupIds	RBAC Scope	Permitted users list	Note
1	<code>["none"]</code>	<code>[]</code>	Empty	No users have access	The values <code>["none"]</code> and <code>[]</code> behave exactly the same
2	<code>["none"]</code>	<code>[]</code>	<code>scope/to/container1</code>	Users with RBAC permissions to container1	The value of "none" doesn't block access

Document #	userIds	groupIds	RBAC Scope	Permitted users list	Note
					by matching other ACL fields
3	["none"]	["group1", "group2"]	Empty	Members of group1 or group2	
4	["all"]	["none"]	Empty	Any user	Any querying user matches the ACL filter "all", so all users have access
5	["all"]	["group1", "group2"]	scope/to/container1	Any user	Since all users match the "all" filter for userID, the groupID and RBAC filters don't have any impact
6	["user1", "user2"]	["group1"]	Empty	User1, user2, or any member of group1	
7	["user1", "user2"]	[]	Empty	User1, user2, or any user with RBAC permissions to container1	

See also

- [Connect to Azure AI Search using roles](#)
- [Query-Time ACL and RBAC enforcement](#)
- [azure-search-python-samples/Quickstart-Document-Permissions-Push-API ↗](#)

Use an ADLS Gen2 indexer to ingest permission metadata and filter search results based on user access rights

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

The permission model in Azure Data Lake Storage (ADLS) Gen2 allows for per-user access to specific directories or files. Preview APIs in Azure AI Search now support ingestion of user permissions alongside document ingestion so that you can use those permissions to control access to search results. If a user lacks permissions on a specific directory or file in ADLS Gen2, that user doesn't have access to the corresponding documents in Azure AI Search results.

- 2025-05-01-preview and later, ADLS Gen2 permissions can be ingested using the [ADLS Gen2 indexer](#).
- 2025-11-01-preview provides equivalent support for [ADLS Gen2 blob knowledge sources](#) in Azure Storage.

You can use the push APIs to upload and index content and permission metadata manually, or you can use an indexer or knowledge source to automate data ingestion.

This article focuses on the indexing automation approaches, built on this foundation:

- The [ADLS Gen2 access control model](#) that provides [Access control lists \(ACLs\)](#) and [Role-based access control \(Azure RBAC\)](#). There's no support for Attribute-based access control (Azure ABAC).
- An [ADLS Gen2 indexer](#) or [ADLS Gen2 blob knowledge source](#) that retrieves and ingests data and metadata, including permission filters. To get permission filter support, use the latest preview REST API or a preview package of an Azure SDK that supports the feature.
- An [index in Azure AI Search](#) containing the ingested documents and corresponding permissions. Permission metadata is stored as fields in the index. To set up [queries that respect the permission filters](#), use the latest preview REST API or a preview package of an Azure SDK that supports the feature.

This functionality helps align [document-level permissions](#) in the search index with the access controls defined in ADLS Gen2, allowing users to retrieve content in a way that reflects their existing permissions.

This article supplements [Index data from ADLS Gen2](#) and [ADLS Gen2 blob knowledge sources](#) with information that's specific to ingesting permissions alongside document content into an Azure AI Search index.

Prerequisites

- [Microsoft Entra ID authentication and authorization](#). Services and apps must be in the same tenant. Users can be in different tenants as long as all of the tenants are Microsoft Entra ID. Role assignments are used for each authenticated connection.
- Azure AI Search, any region, but you must have a billable tier (basic and higher) for managed identity support. The search service must be [configured for role-based access](#) and it must [have a managed identity \(either system or user\)](#).
- ADLS Gen2 blobs in a hierarchical namespace, with user permissions granted through ACLs or roles.

Limitations

- [Limits on Azure role assignments and ACL entries](#) in ADLS Gen2 impose a maximum number of role assignments and ACL entries.
- The `owning users`, `owning groups`, `Other (all)`, [ACL identities categories](#) aren't supported during public preview. Use `named users` and `named groups` assignments instead.
- The following indexer features don't support permission inheritance in indexed documents originating from ADLS Gen2. If you use any of these features in a skillset or indexer, document-level permissions aren't included in the indexed content.
 - [Custom Web API skill](#)
 - [GenAI Prompt skill](#)
 - [Knowledge store](#)
 - [Indexer enrichment cache](#)
 - [Debug sessions](#)
- This functionality is currently not supported in the Azure portal.

Support for the permission model

This section compares document-level access control features between ADLS Gen2 and Azure AI Search. It explains which Azure Data Lake Storage (ADLS) Gen2 access control mechanisms AI Search supports or maps. This helps you understand how permissions are enforced at the document level.

[+] [Expand table](#)

ADLS Gen2 Feature	Description	Supported	Notes
RBAC	Coarse-grained access at container level	Yes	AI Search honors RBAC for access to all documents in the entire container.
ABAC	Attribute-based conditions on top of RBAC	No	AI Search doesn't evaluate ABAC conditions for document-level access.
ACL	Fine-grained permissions at directory/file (document) level	Yes	AI Search uses document-level ACLs for permission filters .
Security Groups	Group-based permission assignments	Yes	Supported if security groups are mapped inside the document-level ACL.

About ACL hierarchical permissions

Indexers and knowledge sources can retrieve ACL assignments from the specified container and all directories leading to each file by following the ADLS Gen2 [hierarchical access evaluation flow](#). The final effective access lists for each file are computed and the different access categories are indexed into the corresponding index fields.

For example, in [ADLS Gen2 common scenarios related to permissions](#) as the file path /Oregon/Portland/Data.txt.

[+] [Expand table](#)

Operation	/	Oregon/	Portland/	Data.txt
Read Data.txt	--X	--X	--X	R--

The indexer or knowledge source collects ACLs from each container and directory. It then determines effective access at lower levels and continues until it resolves permissions for every file.

txt

```
/ assigned access vs Oregon/ assigned access  
=> Oregon/ effective access vs Portland/ assigned access  
=> Portland/ effective access vs Data.txt assigned access  
=> Data.txt effective access
```

Configure ADLS Gen2

An indexer or knowledge source can retrieve ACLs on a storage account if the following criteria are met. For more information about ACL assignments, see [ADLS Gen2 ACL assignments](#).

Authorization

For indexing, your search service identity must have **Storage Blob Data Reader** permission.

If you're testing locally, you should also have a **Storage Blob Data Reader** role assignment. For more information, see [Connect to Azure Storage using a managed identity](#).

Root container permissions:

1. Assign all `Group` and `User` sets (security principals) at the root container `/` with `Read` and `Execute` permissions.
2. Ensure both `Read` and `Execute` are added as "Default permissions" so they propagate to newly created files and directories automatically.

Propagate permissions down the file hierarchy

Although new directories and files inherit permissions, existing directories and files don't automatically inherit these assignments.

Use the ADLS Gen2 tool to [apply ACLs recursively](#) for assignments propagation on existing content. This tool propagates the root container's ACL assignments to all underlying directories and files.

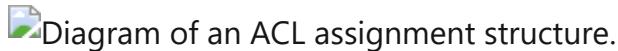
Remove excess permissions

After applying ACLs recursively, review permissions for each directory and file.

Remove any `Group` or `User` sets that shouldn't have access to specific directories or files. For example, remove `User2` on folder `Portland/`, and for folder `Idaho` remove `Group2` and `User2` from its assignments, and so on.

Sample ACL assignments structure

Here's a diagram of the ACL assignment structure for the [fictitious directory hierarchy](#) in the ADLS Gen2 documentation.



Updating ACL assignments over time

Over time, as any new ACL assignments are added or modified, repeat the above steps to ensure proper propagation and permissions alignment. Updated permissions in ADLS Gen2 are updated in the search index when you re-ingest the content using the indexer or knowledge source.

Configure Azure AI Search

Recall that the search service must have:

- [Role-based access enabled](#)
- [Managed identity configured](#)

Authorization

For indexing, the client issuing the API call must have **Search Service Contributor** permission to create objects, **Search Index Data Contributor** permission to perform data import, and **Search Index Data Reader** to query an index.

If you're testing locally, you should have the same role assignments. For more information, see [Connect to Azure AI Search using roles](#).

Configure a knowledge source

If you're using a knowledge source, definitions in the knowledge source are used to generate a full indexing pipeline (indexer, data source, and index). ACL assignments are detected and automatically included in the generated index. There's no need to modify any of the generated objects if you want permission inheritance in your indexed content.

Key points about the configuration that make it work for this scenario:

- `isADLSGen2` is set to true, meeting the data source requirement for this scenario.
- `ingestionPermissionOptions` specifies user and group IDs.

- `disableImageVerbalization` is set to true because the GenAI Prompt skill that backs this experience isn't currently supported in ADLS Gen2 permission inheritance.

HTTP

```
# Create / Update Azure Blob Knowledge Source
###  

PUT {{url}}/knowledgesources/azure-blob-ks?api-version=2025-11-01-preview  

api-key: {{key}}  

Content-Type: application/json  
  

{  

  "name": "azure-blob-ks",  

  "kind": "azureBlob",  

  "description": "A sample azure blob knowledge source",  

  "azureBlobParameters": {  

    "connectionString": "{{blob-connection-string}}",  

    "containerName": "blobcontainer",  

    "FolderPath": null,  

    "isADLSSGen2": true,  

    "ingestionParameters": {  

      "identity": null,  

      "embeddingModel": {  

        "kind": "azureOpenAI",  

        "azureOpenAIParameters": {  

          "deploymentId": "text-embedding-3-large",  

          "modelName": "text-embedding-3-large",  

          "resourceUri": "{{aoai-endpoint}}",  

          "apiKey": "{{aoai-key}}"  

        }  

      },  

      "chatCompletionModel": null,  

      "disableImageVerbalization": true,  

      "ingestionSchedule": null,  

      "ingestionPermissionOptions": [  

        "userIds", "groupIds"  

      ],  

      "contentExtractionMode": "minimal",  

      "aiServices": {  

        "uri": "{{ai-endpoint}}",  

        "apiKey": "{{ai-key}}"  

      }  

    }  

  }  

}  

}###
```

Configure indexer-based indexing

If you're using an indexer, configure it, the data source, and the index to pull permission metadata from ADLS Gen2 blobs.

Create the data source

This section supplements [Index data from ADLS Gen2](#) with information that's specific to ingesting permissions alongside document content into an Azure AI Search index.

- Data Source type must be `adlsgen2`.
- Data source must have `indexerPermissionOptions` with `userIds`, `groupIds` and/or `rbacScope`.
 - For `rbacScope`, configure the [connection string](#) with managed identity format.
 - For connection strings using a [user-assigned managed identity](#), you must also specify the `identity` property.

JSON example with system managed identity:

JSON

```
{  
  "name" : "my-adlsgen2-acl-datasource",  
  "type": "adlsgen2",  
  "indexerPermissionOptions": ["userIds", "groupIds", "rbacScope"],  
  "credentials": {  
    "connectionString": "ResourceId=/subscriptions/<your subscription  
ID>/resourceGroups/<your resource group  
name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;"  
  },  
  "container": {  
    "name": "<your container name>",  
    "query": "<optional-virtual-directory-name>"  
  }  
}
```

JSON schema example with a user-managed identity in the connection string:

JSON

```
{  
  "name" : "my-adlsgen2-acl-datasource",  
  "type": "adlsgen2",  
  "indexerPermissionOptions": ["userIds", "groupIds", "rbacScope"],  
  "credentials": {  
    "connectionString": "ResourceId=/subscriptions/<your subscription  
ID>/resourceGroups/<your resource group  
name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;"  
  },  
  "container": {  
    "name": "<your container name>",  
    "query": "<optional-virtual-directory-name>"  
  }  
}
```

```

},
"identity": {
"@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
"userAssignedIdentity": "/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{user-assigned-managed-identity-name}"
}
}

```

Create permission fields in the index

In Azure AI Search, make sure your index contains field definitions for the permission metadata. Permission metadata can be indexed when `indexerPermissionOptions` is specified in the data source definition.

Recommended schema attributes for ACL (UserIds, GroupIds) and RBAC Scope:

- User identifier (ID) field with `userIds` permissionFilter value.
- Group IDs filed with `groupIds` permissionFilter value.
- RBAC scope field with `rbacScope` permissionFilter value.
- Property `permissionFilterOption` to enable filtering at querying time.
- Use string fields for permission metadata
- Set `filterable` to true on all fields.

Notice that `retrievable` is false. You can set it true during development to verify permissions are present, but remember to set to back to false before deploying to a production environment.

JSON schema example:

JSON

```
{
...
"fields": [
...
  { "name": "UserIds", "type": "Collection(Edm.String)", "permissionFilter": "userIds", "filterable": true, "retrievable": false },
  { "name": "GroupIds", "type": "Collection(Edm.String)", "permissionFilter": "groupIds", "filterable": true, "retrievable": false },
  { "name": "RbacScope", "type": "Edm.String", "permissionFilter": "rbacScope", "filterable": true, "retrievable": false }
],
"permissionFilterOption": "enabled"
}
```

Configure the indexer

Field mappings within an indexer set the data path to fields in an index. Target and destination fields that vary by name or data type require an explicit field mapping. The following metadata fields in ADLS Gen2 might need field mappings if you vary the field name:

- `metadata_user_ids` (`Collection(Edm.String)`) - the ACL user IDs list.
- `metadata_group_ids` (`Collection(Edm.String)`) - the ACL group IDs list.
- `metadata_rbac_scope` (`Edm.String`) - the container RBAC scope.

Specify `fieldMappings` in the indexer to route the permission metadata to target fields during indexing.

JSON schema example:

JSON

```
{  
  ...  
  "fieldMappings": [  
    { "sourceFieldName": "metadata_user_ids", "targetFieldName": "UserIds" },  
    { "sourceFieldName": "metadata_group_ids", "targetFieldName": "GroupIds" },  
    { "sourceFieldName": "metadata_rbac_scope", "targetFieldName": "RbacScope" }  
  ]  
}
```

Recommendations and best practices

- Plan the ADLS Gen2 folder structure carefully before creating any folders.
- Organize identities into groups and use groups whenever possible, rather than granting access directly to individual users. Continuously adding individual users instead of applying groups increases the number of access control entries that must be tracked and evaluated. Not following this best practice can lead to more frequent security metadata updates required to the index as this metadata changes, causing increased delays and inefficiencies in the refresh process.

Synchronize permissions between indexed and source content

Enabling ACL or RBAC enrichment on an indexer works automatically only in two situations:

- **The very first full indexer run / data crawl:** all permission metadata that exists at that moment for each document is captured.
- **Brand-new documents added after ACL/RBAC support is enabled:** their ACL/RBAC information is ingested along with their content.

If you change document permissions, such as adding a user to an ACL or updating a role assignment, the change doesn't appear in the search index unless you tell the indexer to crawl the document's permission metadata again.

Choose one of the following mechanisms, depending on how many items changed:

 Expand table

Scope of your change	Best trigger	What gets refreshed on the next run
A single blob or just a handful	Update the blob's <code>Last-Modified</code> timestamp in storage (touch the file)	Document content and ACL/RBAC metadata
Dozens to thousands of blobs	Call /resetdocs (preview) and list the affected document keys.	Document content and ACL/RBAC metadata
Entire data source	Call /resync (preview) with the permissions option.	Only ACL/RBAC metadata (content is left untouched)

Resetdocs (preview) API example:

HTTP

```
POST https://{{service}}.search.windows.net/indexers/{{indexer}}/resetdocs?api-version=2025-11-01-preview
{
  "documentKeys": [
    "1001",
    "4452"
  ]
}
```

Resync (preview) API example:

HTTP

```
POST https://{{service}}.search.windows.net/indexers/{{indexer}}/resync?api-version=2025-11-01-preview
{
  "options": [
    "permissions"
  ]
}
```

]
}

ⓘ Important

If you change permissions on indexed documents and don't trigger one of the mechanisms above, the search index continues serving outdated ACL or RBAC data. New documents continue to be indexed automatically; no manual trigger is needed for them.

Deletion tracking

To manage blob deletion effectively, make sure [deletion tracking](#) is enabled before your indexer runs for the first time. This feature lets the system detect deleted blobs in your source and remove them from the index.

See also

- [Connect to Azure AI Search using roles](#)
- [Query-time ACL and RBAC enforcement](#)
- [azure-search-python-samples/Quickstart-Document-Permissions-Push-API ↗](#)

Last updated on 11/19/2025

Use a Blob indexer or knowledge source to ingest RBAC scopes metadata

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Azure Storage allows for role-based access on containers in blob storage, where roles like **Storage Blob Data Reader** or **Storage Blob Data Contributor** determine whether someone has access to content. Preview APIs in Azure AI Search now support ingestion of user permissions alongside document ingestion so that you can use those permissions to control access to search results. If a user lacks permissions on a specific directory or file in Azure Storage, that user doesn't have access to the corresponding documents in Azure AI Search results, even if you personally have a **Search Index Data Reader** assignment *on the index*.

- 2025-05-01-preview and later, RBAC scopes metadata can be ingested using the [Blob indexer](#).
- 2025-11-01-preview provides equivalent support for [Blob knowledge sources](#) in Azure Storage.

RBAC scope is set at the container level and flows to all blobs (documents) through permission inheritance. RBAC scope is captured during indexing as permission metadata. You can use the push APIs to upload and index content and permission metadata manually (see [Indexing Permissions using the push REST API](#)), or you can use an indexer or knowledge source to automate data ingestion. This article focuses on indexing automation.

At query time, the identity of the caller is included in the request header via the `x-ms-query-source-authorization` parameter. The identity must match the permission metadata on documents if the user is to see the search results.

This article focuses on the indexing automation approaches, built on this foundation:

- [Azure Storage blobs secured using role-based access control \(Azure RBAC\)](#). There's no support for Attribute-based access control (Azure ABAC).
- [Azure Blob indexer](#) or a [Blob knowledge source](#) that retrieves and ingests data and metadata, including permission filters. To get permission filter support, use the latest preview REST API or a preview package of an Azure SDK that supports the feature.

- An index in Azure AI Search containing the ingested documents and corresponding permissions. Permission metadata is stored as fields in the index.
- A query that uses permission filters. To set up queries that respect the permission filters, use the latest preview REST API or a preview package of an Azure SDK that supports the feature.

Prerequisites

- Microsoft Entra ID authentication and authorization. Services and apps must be in the same tenant. Users can be in different tenants as long as all of the tenants are Microsoft Entra ID. Role assignments are used for each authenticated connection.
- Azure AI Search, any region, but you must have a billable tier (basic and higher) for managed identity support. The search service must be [configured for role-based access](#) and it must [have a managed identity \(either system or user\)](#).
- Azure Storage, Standard performance (general-purpose v2), on hot, cool, and cold access tiers, with RBAC-secured containers or blobs.
- You should understand how indexers and knowledge sources work and how to create an index. This article explains the configuration settings for the data source and indexer, but doesn't provide steps for creating the index. For more information about indexes designed for permission filters, see [Create an index with permission filter fields](#).
- This functionality is currently not supported in the Azure portal, this includes Permission filters created through the [Import wizards](#). Use a programmatic approach to create or modify existing objects for document-level access.

Configure Blob storage

Verify your blob container uses role-based access.

1. Sign in to the Azure portal and find your storage account.
2. Expand **containers** and select the container that has the blobs you want to index.
3. Select **Access Control (IAM)** to check role assignments. Users and groups with **Storage Blob Data Reader** or **Storage Blob Data Contributor** will have access to search documents in the index after the container is indexed.

Authorization

For indexer execution, your search service identity must have **Storage Blob Data Reader** permission. For more information, see [Connect to Azure Storage using a managed identity](#).

Configure Azure AI Search

Recall that the search service must have:

- Role-based access enabled
- Managed identity configured

Authorization

For indexer execution, the client issuing the API call must have **Search Service Contributor** permission to create objects, **Search Index Data Contributor** permission to perform data import, and **Search Index Data Reader** to query an index see [Connect to Azure AI Search using roles](#).

Configure a knowledge source

If you're using a knowledge source, definitions in the knowledge source are used to generate a full indexing pipeline (indexer, data source, and index). RBAC scope is detected and automatically included in the generated index. There's no need to modify any of the generated objects if you want permission inheritance in your indexed content.

Key points about the configuration that make it work for this scenario:

- `isADLSGen2` is set to false, which means the data source is Azure Blob storage.
- `ingestionPermissionOptions` specifies `rbacScope`.

HTTP

```
# Create / Update Azure Blob Knowledge Source
###
PUT {{url}}/knowledgesources/azure-blob-ks?api-version=2025-11-01-preview
api-key: {{key}}
Content-Type: application/json

{
    "name": "azure-blob-ks",
    "kind": "azureBlob",
    "description": "A sample azure blob knowledge source",
    "azureBlobParameters": {
        "connectionString": "{{blob-connection-string}}",
        "containerName": "blobcontainer",
        "FolderPath": null,
```

```
        "isADLSGen2": false,
        "ingestionParameters": {
            "identity": null,
            "embeddingModel": {
                "kind": "azureOpenAI",
                "azureOpenAIParameters": {
                    "deploymentId": "text-embedding-3-large",
                    "modelName": "text-embedding-3-large",
                    "resourceUri": "{{aoai-endpoint}}",
                    "apiKey": "{{aoai-key}}"
                }
            },
            "chatCompletionModel": null,
            "disableImageVerbalization": true,
            "ingestionSchedule": null,
            "ingestionPermissionOptions": ["rbacScope"],
            "contentExtractionMode": "minimal",
            "aiServices": {
                "uri": "{{ai-endpoint}}",
                "apiKey": "{{ai-key}}"
            }
        }
    }
}
```

Configure indexer-based indexing

If you're using an indexer, configure it, the data source, and the index to pull permission metadata from blobs.

Create the data source

- Data Source type must be `azureblob`.
- Data source parsing mode must be the default.
- Data source must have `indexerPermissionOptions` with `rbacScope`.
 - For `rbacScope`, configure the [connection string](#) with managed identity format.
 - For connection strings using a [user-assigned managed identity](#), you must also specify the `identity` property.

JSON example with system managed identity and `indexerPermissionOptions`:

JSON

```
{
  "name" : "my-blob-datasource",
  "type": "azureblob",
  "indexerPermissionOptions": ["rbacScope"],
  "credentials": {
    "connectionString": "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;"
  },
  "container": {
    "name": "<your-container-name>",
    "query": "<optional-query-used-for-selecting-specific-blobs>"
  }
}
```

JSON schema example with a user-managed identity in the connection string:

JSON

```
{
  "name" : "my-blob-datasource",
  "type": "azureblob",
  "indexerPermissionOptions": ["rbacScope"],
  "credentials": {
    "connectionString": "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/";
  },
  "container": {
    "name": "<your-container-name>",
    "query": "<optional-query-used-for-selecting-specific-blobs>"
  },
  "identity": {
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
    "userAssignedIdentity": "/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{user-assigned-managed-identity-name}"
  }
}
```

Create permission fields in the index

In Azure AI Search, make sure your index contains field definitions for the permission metadata. Permission metadata can be indexed when `indexerPermissionOptions` is specified in the data source definition.

Recommended schema attributes RBAC Scope:

- RBAC scope field with `rbacScope` permissionFilter value.

- Property `permissionFilterOption` to enable filtering at querying time.
- Use string fields for permission metadata
- Set `filterable` to true on all fields.

Notice that `retrievable` is false. You can set it true during development to verify permissions are present, but remember to set it back to false before deploying to a production environment so that security principal identities aren't visible in results.

JSON schema example:

```
JSON
{
  ...
  "fields": [
    ...
    {
      "name": "RbacScope",
      "type": "Edm.String",
      "permissionFilter": "rbacScope",
      "filterable": true,
      "retrievable": false
    }
  ],
  "permissionFilterOption": "enabled"
}
```

Configure the indexer

Field mappings within an indexer set the data path to fields in an index. Target and destination fields that vary by name or data type require an explicit field mapping. The following metadata fields in Azure Blob Storage might need field mappings if you vary the field name:

- `metadata_rbac_scope` (`Edm.String`) - the container RBAC scope.

Specify `fieldMappings` in the indexer to route the permission metadata to target fields during indexing.

JSON schema example:

```
JSON
{
  ...
  "fieldMappings": [
    { "sourceFieldName": "metadata_rbac_scope", "targetFieldName": "RbacScope" }
  ]
}
```

```
]  
}
```

Run the indexer

Once your indexer, data source, and index are configured, run the indexer to set the process in motion. If there's a problem with configuration or permissions, those problems will surface in this step.

By default, an indexer runs as soon as you post it to a search service, but if the indexer configuration includes `disabled` set to true, the indexer is posted in a disabled state so that you can run the indexer manually.

We recommend [running the indexer from the Azure portal](#) so that you can monitor status and messages.

Assuming no errors, the index is now populated and you can move forward with [queries and testing](#).

Deletion tracking

To effectively manage blob deletion, ensure that you have enabled [deletion tracking](#) before your indexer runs for the first time. This feature allows the system to detect deleted blobs from your source and delete the corresponding content from the index.

See also

- [Connect to Azure AI Search using roles](#)
- [Query-time ACL and RBAC enforcement](#)
- [azure-search-python-samples/Quickstart-Document-Permissions-Push-API ↗](#)
- [Search over Azure Blob Storage content](#)
- [Configure a blob indexer](#)

Use a SharePoint indexer to ingest permission metadata and filter search results based on user access rights

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

This article explains how to ingest an Access Control List (ACL) alongside other content from SharePoint in Microsoft 365 using an Azure AI Search indexer. Permissions from SharePoint are preserved as permission metadata for each indexed document. When users query an index containing content from SharePoint, their search results consist of only those documents for which they have permission to access.

i Important

For scenarios that require the full SharePoint permissions model, sensitivity labels, and out-of-the-box security trimming, use a [remote SharePoint knowledge source](#). This approach calls SharePoint directly via the [Copilot retrieval API](#) so governance remains fully in SharePoint and query results automatically respect all applicable permissions and labels.

Prerequisites

- [Azure AI Search](#) (Basic or higher).
- SharePoint in Microsoft 365 sites, libraries, folders, and files with configured permissions.
- Follow all configuration steps mentioned in the [SharePoint indexer documentation](#). Make sure that you apply the specific requirements in this document for ACL ingestion configuration.
- Configure [Application permissions](#) with `Files.Read.All` and `Sites.FullControl.All` (or `Sites.Selected` instead of `Sites.FullControl.All`), to index only the content and permissions of specific sites. Then, grant the application full control permissions for just those selected sites.

Limitations

- During public preview, this functionality applies to initial ingestion only: ACLs are captured on the first ingestion of each file. If permissions change in the source, you must [explicitly reindex those documents or their respective ACLs](#).
- Not supported in this preview:
 - [SharePoint Information Management policies](#) applicable to user access.
 - Document [shareable](#) "Anyone links" or "People in your organization links". Only "specific people links" sync are supported.
 - [SharePoint groups](#) that can't be resolved to Microsoft Entra groups (such as Owners, Members, Visitors groups).
 - Azure portal is out of support during preview; use REST API version 2025-11-01-preview or SDK preview packages.
 - This feature must not be tested in combination with [sensitivity labels preservation and honoring](#) feature at this time. Both features must be tested on different indexers and indexes accordingly, since their coexistence is not supported currently.

Support for the SharePoint permission model

This preview supports only basic ACLs for documents, as shown in the following table. The SharePoint indexer doesn't support lists ingestion, so it excludes lists permissions.

[] Expand table

SharePoint Feature	Description	Supported	Notes
Site & library inheritance	Site → library → folder → file.	✓	Evaluated at ingestion; effective ACLs computed per file.
Folder & file unique ACLs	Item-level access.	✓	Included when present at first ingestion.
Microsoft Entra (M365/Security) Groups	Group-based access.	✓	Group IDs included when resolvable to Entra identifier (ID).
SharePoint site groups	Owners/Members/Visitors.	⚠ Partial	Included only when resolvable to Entra group ID.
Shareable "Anyone links" or "People in your organization links"	Org-wide or public access.	✗	Not supported in preview.

SharePoint Feature	Description	Supported	Notes
External/guest users	Access for guests.	✗	Not supported.
Information Management policies	Policies to define specific permissions requirements.	✗	Not supported in preview.
Purview sensitivity labels	Document-level security for privacy, categorization, permissions, and encryption	✗	Supported via a separate feature: preserving and honoring sensitivity labels and not to be tested in the same indexer/index as this ACL feature at this time.

How hierarchical permissions are evaluated

SharePoint permissions inherit the hierarchy of Site → Library → Folder → File, unless inheritance is broken.

During ingestion, the indexer gathers user and group identifiers (ID) at each level and computes the effective ACL for each file.

Configure your search service for ACL ingestion and honoring at query time

These steps configure your search service for ACL ingestion and enable ACL honoring at query time.

1. Data source configuration

Set `indexerPermissionOptions` in the [data source definition](#) to allow indexing of `userIds` and `groupIds` from SharePoint documents.

```
{
  "name": "my-sharepoint-acl-datasource",
  "type": "sharepoint",
  "indexerPermissionOptions": ["userIds", "groupIds"],
  "credentials": {
    "connectionString": "<connection-string>;"
  },
  "container": {
    "name": "<library-name>",
    "query": "<optional-folder-path>"}
```

```
}
```

2. Add permission fields to the index definition

Add fields to your [index schema definition](#) to store ACLs and support query-time filtering.

```
{
  "fields": [
    { "name": "UserIds", "type": "Collection(Edm.String)", "permissionFilter": "userIds", "filterable": true, "retrievable": false },
    { "name": "GroupIds", "type": "Collection(Edm.String)", "permissionFilter": "groupIds", "filterable": true, "retrievable": false }
  ],
  "permissionFilterOption": "enabled"
}
```

Set `retrievable` attribute to `true` only during development to verify values. You can change `retrievable` from `true` to `false` with no index rebuild requirement.

3. Configure index projections in your skillset (if applicable)

If your indexer uses a [skillset](#) with data chunking, such as a [split skill](#) when enabling [integrated vectorization](#), make sure to map ACL properties to each chunk using [index projections](#).

```
PUT https://{{service}}.search.windows.net/skillsets/{{skillset}}?api-version=2025-11-01-preview
{
  "name": "my-skillset",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
      "name": "#split",
      "context": "/document",
      "inputs": [{ "name": "text", "source": "/document/content" }],
      "outputs": [{ "name": "textItems", "targetName": "chunks" }]
    }
    // ... (other skills such as embeddings, entity recognition, etc.)
  ],
  "indexProjections": {
    "selectors": [
      {
        "targetIndexName": "chunks-index",
        "parentKeyFieldName": "parentId",           // must exist in target index
        "sourceContext": "/document/chunks/*",       // match your split output path
      }
    ]
  }
}
```

```

    "mappings": [
        { "name": "chunkId",           "source": "/document/chunks/*/id" },      // if you create an id per chunk
        { "name": "content",          "source": "/document/chunks/*/text" },      // chunk text
        { "name": "parentId",         "source": "/document/id" },                // parent doc id
        { "name": "UserIds",          "source": "/document/metadata_user_ids" } // <-- parent → child
        { "name": "GroupIds",         "source": "/document/metadata_group_ids" } // <-- parent → child
    ],
    "parameters": {
        "projectionMode": "skipIndexingParentDocuments"
    }
}
}

```

4. Configure the indexer field mappings for ACLs

Besides your required [indexer configuration](#), map raw metadata ACL fields from SharePoint to your index fields.

```
{
  "fieldMappings": [
    { "sourceFieldName": "metadata_user_ids", "targetFieldName": "UserIds" },
    { "sourceFieldName": "metadata_group_ids", "targetFieldName": "GroupIds" }
  ]
}
```

Synchronize permissions between indexed and source content

During public preview when the configuration is completed, and ACLs are captured during the first indexer run and for new files only. To pick up later changes:

 Expand table

Change Scope	Recommended	Trigger	What refreshes
Single/few files	Update	LastModified	Content and ACLs

Change Scope	Recommended	Trigger	What refreshes
Many items	Update	/resetdocs (preview) with document keys	Content and ACLs
Entire site/library (as defined in the data source configuration)	Update	/resync (preview) with permissions	Only ACLs (no content refresh)

Reset specific documents

You can [reset specific documents](#) to fully ingest again content and ACLs.

```
POST https://{{service}}.search.windows.net/indexers/{{indexer}}/resetdocs?api-version=2025-11-01-preview
{
  "documentKeys": ["doc123", "doc456"]
}
```

Resync ACLs across the full data source

You can [resync the full data set ACL content](#) after initial ingestion.

```
POST https://{{service}}.search.windows.net/indexers/{{indexer}}/resync?api-version=2025-11-01-preview
{
  "options": ["permissions"]
}
```

i Important

If you change SharePoint permissions without triggering an update mechanism, the index serves stale ACL data for previously ingested files.

After indexing your data and ACLs, you can [query the index](#).

See also

[Index SharePoint content in Azure AI Search \(preview\)](#)

[Query-Time ACL enforcement](#)

Last updated on 11/18/2025

Query-time ACL and RBAC enforcement in Azure AI Search

Query-time access control ensures that users only retrieve search results they're authorized to access, based on their identity, group memberships, roles, or attributes. This functionality is essential for secure enterprise search and compliance-driven workflows.

Authorized access depends on permission metadata that's ingested during indexing. For indexer data sources that have built-in access models, such as Azure Data Lake Storage (ADLS) Gen2 and SharePoint in Microsoft 365, an indexer can pull in the permission metadata for each document automatically. For other data sources, you must assemble the document payload yourself, and the payload must include both content and the associated permission metadata. You then use the [push APIs](#) to load the index.

This article explains how to set up queries that use permission metadata to filter results.

Prerequisites

- Permission metadata must be in `filterable` string fields. You won't use the filter in your queries, but the search engine builds a filter internally to exclude unauthorized content.
- Permission metadata must consist of either POSIX-style permissions that identify the level of access and the group or user ID, or the resource ID of the container in ADLS Gen2 if you're using RBAC scope.
- Depending on the data source:
 - For ADLS Gen2 data sources, you must have configured Access Control Lists (ACLs) and/or Azure role-based access control (RBAC) roles at the container level.
 - For Azure Blob data sources, you must have role assignments on the container. You can use a [built-in indexer](#), a [knowledge source](#), or [Push APIs](#) to index permission metadata in your index.
 - For SharePoint data sources, you must have configured Access Control Lists (ACLs). You can use a [built-in SharePoint indexer](#) and configure it with [ACL ingestion capabilities](#).
- Use the [latest preview REST API](#) or a preview package of an Azure SDK to query the index or knowledge source. This API version supports internal queries that filter out unauthorized results.

Limitations

- If ACL evaluation fails (for example, the Graph API is unavailable), the service returns **5xx** and does **not** return a partially filtered result set.
- Document visibility requires both:
 - the calling application's RBAC role (Authorization header)
 - the user identity carried by **x-ms-query-source-authorization**
- Initial ACL-based queries may experience higher latency compared to subsequent requests, due to caching and permission resolution overhead.

ACL entry limits per data source

Access Control List (ACL) entry limits define how many distinct permission records can be associated with a file, folder, or item within a connected data source. Each entry represents a single user or group identity and the access rights granted to that identity (for example, Read, Write, or Execute).

The maximum number of ACL entries supported by Azure AI Search functionality varies depending on the data source type:

Azure Data Lake Storage Gen2 (ADLS Gen2): Each file or directory can have up to [32 ACL entries permissions](#). In this context, an entry means a single principal (user or group) with a specific permission set. Example: assigning "Everyone" read access and "Azure users" execute access would count as two ACL entries.

SharePoint in Microsoft 365: SharePoint data source in search supports up to 1,000 permission entries per file. Each entry represents a unique user or group assignment in the item's permission list. This is distinct from the overall [unique permission scopes limits](#) per list or library, which governs how many items can have unique permissions.

These limits determine how granularly Azure AI Search can honor item-level permissions when indexing or filtering search results. If an item exceeds these ACL entry limits, permissions beyond the limit may not be enforced at query time.

How query-time enforcement works

This section lists the order of operations for ACL enforcement at query time. Operations vary depending on whether you use Azure RBAC scope or Microsoft Entra ID group or user IDs.

1. User permissions input

The end-user application includes a query access token as part of the search query request, and that access token is typically the identity of the user. The following table lists the source of the user permissions supported by Azure AI Search for ACL enforcement:

 Expand table

Permission type	Source
userIds	oid from <code>x-ms-query-source-authorization</code> token
groupIds	Group membership fetched using the Microsoft Graph API
rbacScope	Permissions the user from <code>x-ms-query-source-authorization</code> has on a storage container

2. Security filter construction

Internally, Azure AI Search dynamically constructs security filters based on the user permissions provided. These security filters are automatically appended to any filters that might come in with the query if the index has the permission filter option enabled.

For Azure RBAC, permissions are lists of resource ID strings. There must be an Azure role assignment (Storage Blob Data Reader) on the data source that grants access to the security principal token in the authorization header. The filter excludes documents if there's no role assignment for the principal behind the access token on the request.

3. Results filtering

The security filter efficiently matches the userIds, groupIds, and rbacScope from the request against each list of ACLs in every document in the search index to limit the results returned to ones the user has access to. It's important to note that each filter is applied independently and a document is considered authorized if any filter succeeds. For example, if a user has access to a document through userIds but not through groupIds, the document is still considered valid and returned to the user.

Query example

Here's an example of a query request from [sample code](#). The query token is passed in the request header. The query token is the personal access token of a user or a group identity behind the request.

HTTP

```
POST {{endpoint}}/indexes/stateparks/docs/search?api-version=2025-11-01-preview
Authorization: Bearer {{query-token}}
x-ms-query-source-authorization: {{query-token}}
Content-Type: application/json

{
    "search": "*",
    "select": "name,description,location,GroupIds",
    "orderby": "name asc"
}
```

! Note

If the query token is omitted, only public documents accessible to everyone are returned in the query request.

Elevated permissions for investigating incorrect results

! Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Debugging queries that include permission metadata can be problematic because search results are specific to each user. As a developer or administrator, you might need elevated permissions to return results regardless of the permission metadata so that you can investigate problems with queries returning unauthorized content.

To investigate, you must be able to:

- View the set of documents that the end user is able to view based on that user's permissions.
- View all documents in the index to investigate why some might not be visible to the end user.

You can accomplish these tasks by adding a custom header, `x-ms-enable-elevated-read: true`, to a query.

Permissions for elevated-read requests

Currently, you must [create a custom role](#) to run queries with elevated permissions. Add the `Microsoft.Search/searchServices/indexes/contentSecurity/elevatedOperations/read` permission to run the queries.

Add an elevated-read header to a query

The following example is a query request against a search index.

HTTP

```
POST {endpoint}/indexes('{indexName}')/search.post.search?api-version=2025-11-01-preview
Authorization: Bearer {AUTH_TOKEN}
x-ms-query-source-authorization: Bearer {TOKEN}
x-ms-enable-elevated-read: true

{
    "search": "prototype tests",
    "select": "filename, author, date",
    "count": true
}
```

Important

The `x-ms-enable-elevated-read` header only works on Search POST actions. You can't perform an elevated read query on a [knowledge base retrieve](#) action.

Important ACL functionality behavior change in specific preview API versions

Before REST API version `2025-11-01-preview`, earlier preview versions `2025-05-01-preview` and `2025-08-01-preview` returned all documents when using a service API key or authorized Entra roles, even if no user token was provided. Applications that didn't validate the presence of a user token could inadvertently expose results to end users if not implemented correctly or following best practices.

Starting in November 2025, this behavior changed:

- ACL permission filters now apply even when using only service API keys or Entra authentication across all versions that support ACL.
- If the user token is omitted, ACL-protected content isn't returned.

- To view all documents for troubleshooting, you must explicitly include the elevated-read header when using REST API version `2025-11-01-preview`.

This update helps keep content protected when applications don't enforce best practices for token validation.

See also

- [Tutorial: Index ADLS Gen2 permission metadata](#)
- [Indexing ACLs and RBAC using the push API in Azure AI Search](#)
- [Use an ADLS Gen2 indexer to ingest permission metadata and filter search results based on user access rights](#)
- [Use a Blob indexer to ingest RBAC scopes metadata](#)
- [Use a SharePoint indexer to ingest permission metadata and filter search results based on user access rights](#)

Last updated on 12/01/2025

Use Azure AI Search indexers to ingest Microsoft Purview sensitivity labels and enforce document-level security

ⓘ Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see

[Supplemental Terms of Use for Microsoft Azure Previews](#).

Azure AI Search now supports automatic extraction of [Microsoft Purview sensitivity labels](#) at document-level during indexing, with label-based access control enforced at query time. Available in public preview, this feature enables organizations to align search experiences with existing [information protection policies](#) defined in Microsoft Purview.

With sensitivity label indexing, Azure AI Search extracts and stores metadata that describes each document's sensitivity level. It also enforces label-based access control, ensuring that only authorized users can view or retrieve labeled content in search results.

This functionality is available for the following data sources:

- [Azure Blob Storage](#)
- [Azure Data Lake Storage Gen2](#)
- [SharePoint in Microsoft 365 \(Preview\)](#)
- [Microsoft OneLake](#)

ⓘ Important

The feature is available in all regions except West US2. Use the [REST API version 2025-11-01-preview](#) or a preview SDK to evaluate the feature.

Portal configuration and debug mode for administrators aren't supported at this time.

Policy enforcement

At query time, Azure AI Search evaluates sensitivity labels and enforces [document-level access control](#) in accordance with the user's Microsoft Entra ID token and Purview label policies.

Only users authorized to access content with [READ usage right](#) under a given label can retrieve corresponding documents in search results. There's a delay in how often the labels are pulled from a document after changed.

When configured [on a schedule](#), the indexer pulls new documents and updates from the data source. It captures:

- Newly added documents and their associated sensitivity labels
- Changes in document content
- Updates to sensitivity labels on existing documents

These updates are detected if they occurred since the last indexer run.

Prerequisites

- [Microsoft Purview sensitivity label policies](#) must be configured and [applied to documents](#) before indexing.
- [Global Administrator](#) or [Privileged Role Administrator](#) roles in your Microsoft Entra tenant are required to grant the search service access to Purview APIs and sensitivity labels.
- Both the Azure AI Search service and end users querying the content must belong to the same Microsoft Entra tenant. Guest users and multitenant scenarios aren't supported.
- File types must be included in the [Purview sensitivity labels supported formats list](#) and also be recognized as [Office supported file types](#) by Azure AI Search indexers.

Limitations

- There's a known issue with document deletion and sensitivity labels. When sensitivity labels are enabled for an index, the indexer fails to enumerate the index's documents. As a result, soft delete operations don't run because the indexer can't list the documents that need to be removed. This applies to indexers that support soft delete, including Azure Blob, ADLS Gen2, OneLake, SharePoint.
- Initial release supports REST API version 2025-11-01-preview and associated beta SDK only. There's no portal experience for configuration or management.
- This feature isn't supported when used simultaneously with [ACL-based security filters](#) (currently also in preview). Test each feature independently until Microsoft announces official coexistence support.

- [Autocomplete](#) and [Suggest](#) APIs are disabled for Purview-enabled indexes, as they can't yet enforce label-based access control.
- Guest accounts and cross-tenant queries aren't supported.
- In the initial release, sensitivity label-enabled indexes don't support unlabeled documents and don't return them in query results. This capability will be documented when available.
- The following indexer features don't support documents with sensitivity labels. If you use any of these features in a skillset or indexer, they don't process those documents.
 - Custom Web API skill
 - GenAI Prompt skill
 - Knowledge store
 - Indexer enrichment cache
 - Debug sessions

The following steps must be followed in order to configure sensitivity label synchronization in Azure AI Search.

1. Enable AI Search managed identity

Enable a [system-assigned managed identity](#) for your Azure AI Search service. This identity is required for the indexer to securely access Microsoft Purview and extract label metadata.

2. Enable RBAC on your AI Search service

Enable a [role-based access control \(RBAC\)](#) on your Azure AI Search service. This step is required so content-related operations such as indexing content and query the index succeed. Keep both RBAC and API keys to avoid disrupting operations that rely on API keys.

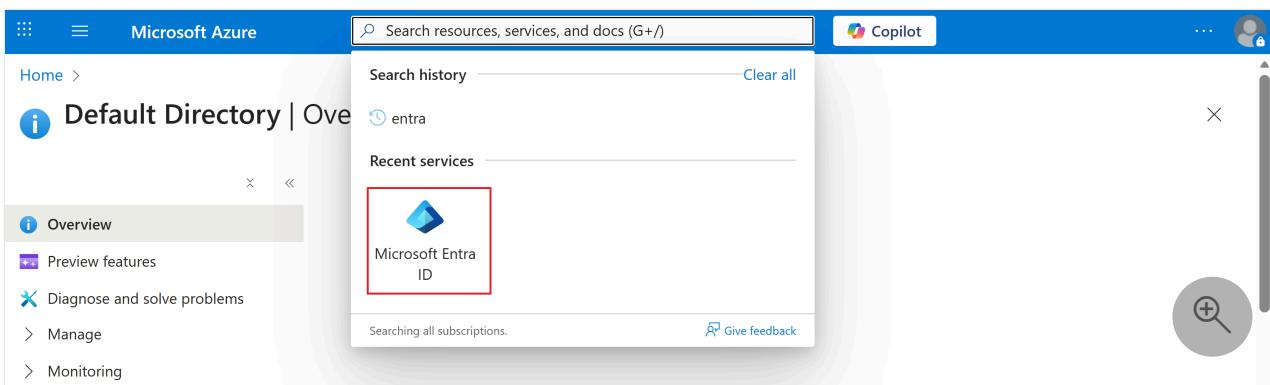
3. Granting access to extract sensitivity labels

Accessing Microsoft Purview sensitivity label metadata involves highly privileged operations, including reading encrypted content and security classifications. To enable this capability in Azure AI Search, you must grant specific roles to the service's managed identity—following your organization's internal governance and approval processes.

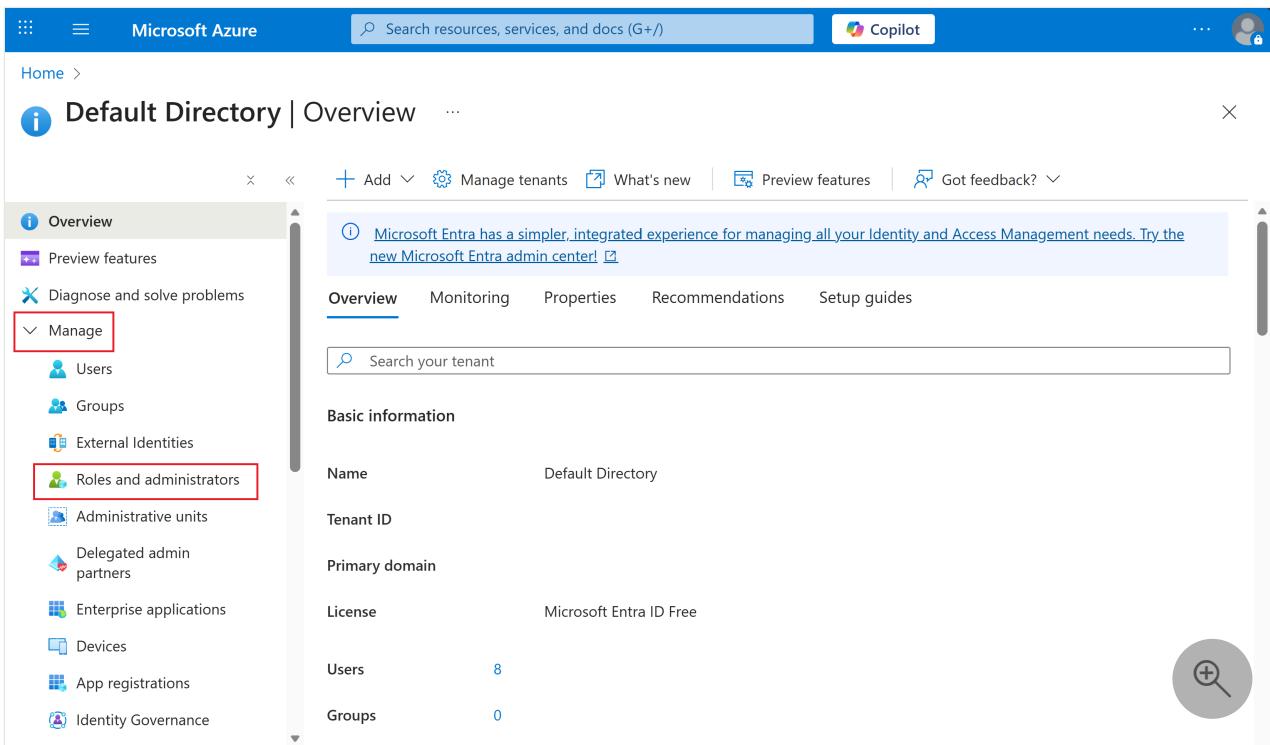
Identify your Global or Privileged Role Administrators

If you need to determine who can authorize permissions for the search service, you can locate active or eligible Global Administrators in your Microsoft Entra tenant.

1. In the [Azure portal](#), search for Microsoft Entra ID.



2. In the left navigation pane, select Manage > Roles and administrators.



3. Search for the Global Administrator or Privileged Role Administrator role and select it.

The screenshot shows the 'Roles and administrators | All roles' page in Microsoft Azure. On the left, a sidebar lists categories like 'All roles', 'Admin units', 'Protected actions', 'Diagnose and solve problems', 'Activity', and 'Troubleshooting + Support'. The main area displays two roles: 'Global Administrator' (selected) and 'Global Reader'. The 'Global Administrator' role is described as 'Can manage all aspects of Microsoft Entra ID and Microsoft services that use Microsoft Entra identities.' It is marked as 'PRIVILEGED' and has 4 assignments. The 'Global Reader' role is described as 'Can read everything that a Global Administrator can, but not update anything.' It is also marked as 'PRIVILEGED' and has 0 assignments. A search bar at the top right is set to 'Global'.

- Under **Eligible assignments** and **Active assignments**, review the list of administrators authorized to run the permissions setup process.

The screenshot shows the 'Global Administrator | Assignments' page. The sidebar includes 'Diagnose and solve problems' and 'Manage' sections, with 'Assignments' selected. The main area shows 'Eligible assignments' tab selected, displaying a table of users assigned to the Global Administrator role. The table columns are Name, UserName, Type, and Scope. Two users are listed: 'azureadmin' and 'demouser', both assigned to 'User' type with 'Directory' scope.

Name	UserName	Type	Scope
<input type="checkbox"/> azureadmin	azureadmin	User	Directory
<input type="checkbox"/> demouser	demouser	User	Directory

Secure Governance Approval

Engage your internal security or compliance teams to review the request. Microsoft recommends following your company's standard governance and security review process before proceeding with any role assignments.

Once approved, a Global Administrator or Privileged Role Administrator must assign the following roles to the Azure AI Search system-assigned managed identity:

- ContentSuperUser – for label and content extraction

- `UnifiedPolicy.Tenant.Read` – for Purview policy and label metadata access

Assign Roles via PowerShell

Your Global Administrator or Privileged Role Administrator should use the following PowerShell script to grant the required permissions. Replace the placeholder values with your actual subscription, resource group, and search service names.

PowerShell

```
Install-Module -Name Az -Scope CurrentUser
Install-Module -Name Microsoft.Entra -AllowClobber
Import-Module Az.Resources
Connect-Entra -Scopes 'Application.ReadWrite.All'

$resourceIdWithManagedIdentity =
"subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.Search/searchServices/<searchServiceName>"
$managedIdentityObjectId = (Get-AzResource -ResourceId
$resourceIdWithManagedIdentity).Identity.PrincipalId

# Microsoft Information Protection (MIP)
$MIPResourceSP = Get-EntraServicePrincipal -Filter "appId eq '870c4f2e-85b6-4d43-bdda-6ed9a579b725'"
New-EntraServicePrincipalAppRoleAssignment -ServicePrincipalId
$managedIdentityObjectId -Principal $managedIdentityObjectId -ResourceId
$MIPResourceSP.Id -Id "8b2071cd-015a-4025-8052-1c0dba2d3f64"

# ARM Service Principal for policy read
$ARMSResourceSP = Get-EntraServicePrincipal -Filter "appId eq '00000012-0000-0000-c000-000000000000'"
New-EntraServicePrincipalAppRoleAssignment -ServicePrincipalId
$managedIdentityObjectId -Principal $managedIdentityObjectId -ResourceId
$ARMSResourceSP.Id -Id "7347eb49-7a1a-43c5-8eac-a5cd1d1c7cf0"
```

The appId roles in the provided PowerShell script are associated to the following Azure roles:

[] Expand table

AppID	Service Principal
870c4f2e-85b6-4d43-bdda-6ed9a579b725	Microsoft Info Protection Sync Service
00000012-0000-0000-c000-000000000000	Azure Resource Manager

4. Configure the index to enable Purview sensitivity label

When sensitivity label support is required, set the `purviewEnabled` property to true in your [index definition](#).

Important

The `purviewEnabled` property must be set to true when the index is created. This setting is permanent and can't be modified later. When `purviewEnabled` is set to true, only RBAC authentication is supported for all document operations APIs. API key access is limited to index schema retrieval (list and get).

```
PUT https://{{service}}.search.windows.net/indexes('{{indexName}}')?api-version=2025-11-01-preview
{
  "purviewEnabled": true,
  "fields": [
    {
      "name": "sensitivityLabel",
      "type": "Edm.String",
      "filterable": true,
      "sensitivityLabel": true,
      "retrievable": true
    }
  ]
}
```

5. Configure the data source

To enable sensitivity label ingestion, configure the [data source](#) with the `indexerPermissionOptions` property set to `["sensitivityLabel"]`.

```
{
  "name": "purview-sensitivity-datasource",
  "type": "azureblob", // < adjust type value according to the data source you are enabling this for: sharepoint, onelake, adlsgen2.
  "indexerPermissionOptions": [ "sensitivityLabel" ],
  "credentials": {
    "connectionString": <your-connection-string>
  },
}
```

```
"container": {  
    "name": "<container-name>"  
}  
}
```

The `indexerPermissionOptions` property instructs the indexer to extract sensitivity label metadata during ingestion and attach it to the indexed document.

6. Configure index projections in your skillset (if applicable)

If your indexer has a [skillset](#) and you're implementing data chunking through [split skill](#), for example, if you have integrated vectorization, you must ensure you also map the sensitivity label to each chunk via [index projections in the skillset](#).

```
PUT https://<service>.search.windows.net/skillsets/<skillset>?api-version=2025-11-01-preview  
{  
    "name": "my-skillset",  
    "skills": [  
        {  
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",  
            "name": "#split",  
            "context": "/document",  
            "inputs": [{ "name": "text", "source": "/document/content" }],  
            "outputs": [{ "name": "textItems", "targetName": "chunks" }]  
        }  
        // ... (other skills such as embeddings, entity recognition, etc.)  
    ],  
    "indexProjections": {  
        "selectors": [  
            {  
                "targetIndexName": "chunks-index",  
                "parentKeyFieldName": "parentId",           // must exist in target index  
                "sourceContext": "/document/chunks/*",      // match your split output path  
                "mappings": [  
                    { "name": "chunkId",           "source": "/document/chunks/*/id" },     //  
                    if you create an id per chunk  
                    { "name": "content",          "source": "/document/chunks/*/text" },   //  
                    chunk text  
                    { "name": "parentId",         "source": "/document/id" },           //  
                    parent doc id  
                    { "name": "sensitivityLabel", "source":  
                        "/document/metadata_sensitivity_label" } // <-- parent → child  
                ]  
            }  
        ],  
        "parameters": {
```

```
        "projectionMode": "skipIndexingParentDocuments"
    }
}
```

7. Configure the indexer

- Define field mappings in your [indexer definition](#) to route extracted label metadata to the index fields. If your data source emits label metadata under a different field name (for example, `metadata_sensitivity_label`), map it explicitly.

```
{
  "fieldMappings": [
    {
      "sourceFieldName": "metadata_sensitivity_label",
      "targetFieldName": "sensitivityLabel"
    }
  ]
}
```

- Sensitivity label updates are indexed automatically when changes to a document's label, content, or metadata are detected during a scheduled indexer run. [Configure the indexer on a recurring schedule](#). The minimum supported interval is every 5 minutes.

Next steps

[How to query a sensitivity labels-enabled index](#)

[Document-level security in Azure AI Search](#)

Last updated on 11/20/2025

Query-Time Microsoft Purview Sensitivity Label Enforcement in Azure AI Search

!Note

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

At query time, Azure AI Search enforces sensitivity label policies defined in [Microsoft Purview](#). These policies include evaluation of [READ usage rights](#) tied to each document. As a result, users can only retrieve documents they are allowed to view.

This capability extends [document-level access control](#) to align with your organization's [information protection and compliance requirements](#) managed in Microsoft Purview.

When Purview sensitivity label indexing is enabled, Azure AI Search checks each document's label metadata during query time. It applies access filters based on Purview policies to return only results the requesting user is allowed to access.

This article explains how query-time sensitivity label enforcement works and how to issue secure search queries.

Prerequisites

Before you can query a sensitivity-label-enabled index, the following conditions must be met:

- You must follow all steps for [Azure AI Search indexers to ingest Microsoft Purview sensitivity labels](#).
- Both the Azure AI Search service and the user issuing the query must belong to the same Microsoft Entra tenant.
- The latest [preview API version 2025-11-01-preview](#) or a compatible beta SDK must be used to query the index.
- Queries must be authenticated using [Azure role-based access control \(RBAC\)](#), not API keys. API key access is restricted to index schema retrieval only when Purview sensitivity labels functionality is enabled.

Limitations

- [Microsoft Entra Guest users](#) and cross-tenant queries aren't supported.
- [Autocomplete](#) and [Suggest](#) APIs are unsupported for Purview-enabled indexes.
- If label evaluation fails (for example, Purview APIs are temporarily unavailable), the service returns 5xx and does **not** return a partial or unfiltered result set.
- [ACL-based security filters](#) aren't supported alongside sensitivity label functionality at this time. Don't enable both at the same time. Once combined usage is supported, it will be documented accordingly.
- The system evaluates labels only as they existed at the time of the last indexer run; recent label changes may not be reflected until the next scheduled reindex.

How query-time sensitivity label enforcement works

When you query an index that includes Microsoft Purview sensitivity labels, Azure AI Search checks the associated Purview policies before returning results. In this way, the query returns only documents that the user token is allowed to access.

1. User identity and application role input

At query time, Azure AI Search validates both:

- The calling application's RBAC role, provided in the `Authorization` header.
- The user identity via token, provided in the `x-ms-query-source-authorization` header.

Both are required to authorize label-based visibility.

 Expand table

Input type	Description	Example source
Application role	Determines whether the calling app has permission to execute queries on the index.	<code>Authorization: Bearer <app-token></code>
User identity	Determines which sensitivity labels the end user is allowed to access.	<code>x-ms-query-source-authorization: Bearer <user-token></code>

2. Sensitivity label evaluation

When a query request is received, Azure AI Search evaluates:

1. The sensitivityLabel field in each indexed document (extracted from Microsoft Purview during ingestion).
2. The user's effective Purview permissions, as defined by Microsoft Entra ID and Purview label policy.

If the user isn't authorized for a document's sensitivity label with extract permissions, that document is excluded from the query results.

 **Note**

Internally, the service builds dynamic access filters similar to RBAC enforcement. These filters aren't user-visible and can't be modified in the query payload.

3. Secure result filtering

Azure AI Search applies the security filter after all user-defined filters and scoring steps. A document is included in the final result set only if:

- The calling application has a valid role assignment (via RBAC), and
- The user identity token represented by `x-ms-query-source-authorization` is valid and permitted to view content with the document's sensitivity label.

If either condition fails, the document is omitted from the results.

Query example

Here's an example of a query request using Microsoft Purview sensitivity label enforcement. The query token is passed in the request headers. Both headers must include valid bearer tokens representing the application and the end user.

HTTP

```
POST {{endpoint}}/indexes/sensitivity-docs/docs/search?api-version=2025-11-01-preview
Authorization: Bearer {{app-query-token}}
x-ms-query-source-authorization: Bearer {{user-query-token}}
Content-Type: application/json

{
  "search": "*",
  "select": "title,summary,sensitivityLabel",
  "orderby": "title asc"
}
```

Sensitivity label handling in Azure AI Search

When Azure AI Search indexes document content with sensitivity labels from sources like SharePoint, Azure Blob, and others, it stores both the content and the label metadata. The search query returns indexed content along with the GUID that identifies the sensitivity label applied to the document, only if the user has data READ access for that document. This GUID uniquely identifies the label but doesn't include human-readable properties such as the label name or associated permissions.

Note that the GUID alone is insufficient for scenarios that include user interface because sensitivity labels often carry other policy controls enforced by [Microsoft Purview Information Protection](#), such as: print permissions or screenshot and screen capture restrictions. Azure AI Search doesn't surface these capabilities.

To display label names and/or enforce UI-specific restrictions, your application must call the Microsoft Purview Information Protection endpoint to retrieve full label metadata and associated permissions.

You can use the GUID returned by Azure AI Search to resolve the label properties and call the [Purview Labels APIs](#) to fetch the label name, description, and policy settings. This [end-to-end demo sample](#) includes code that shows how to call the endpoint from a user interface. It also demonstrates how to extract the label name and expose it as part of the citations used in your RAG applications or agents.

Last updated on 11/21/2025

Configure customer-managed keys for data encryption in Azure AI Search

09/18/2025

Azure AI Search automatically encrypts data at rest with [Microsoft-managed keys](#). If you need another layer of encryption or the ability to revoke keys and shut down access to content, you can use keys that you create and manage in Azure Key Vault. This article explains how to set up customer-managed key (CMK) encryption.

You can store keys using either:

- Azure Key Vault
- Azure Key Vault Managed HSM (Hardware Security Module). An Azure Key Vault Managed HSM is an FIPS 140-2 Level 3 validated HSM. HSM support is new in Azure AI Search. To migrate from Azure Key Vault to HSM, [rotate your keys](#) and choose Managed HSM for storage.

Important

- CMK provides encryption for data at rest. If you need to protect data in use, consider using [confidential computing](#).
- CMK encryption is irreversible. You can rotate keys and change CMK configuration, but index encryption lasts for the lifetime of the index. Post-CMK encryption, an index is only accessible if the search service has access to the key. If you revoke access to the key by deleting or changing role assignment, the index is unusable and the service can't be scaled until the index is deleted or access to the key is restored. If you delete or rotate keys, the most recent key is cached for up to 60 minutes.

CMK encrypted objects

CMK encryption applies to individual objects when they're created. This means you can't encrypt objects that already exist. CMK encryption occurs each time an object is saved to disk, for both data at rest (long-term storage) or temporary cached data (short-term storage). With CMK, the disk never sees unencrypted data.

Objects that can be encrypted include indexes, synonym lists, indexers, data sources, and skillsets. Encryption is computationally expensive to decrypt so only sensitive content is

encrypted.

Encryption is performed over the following content:

- All content within indexes and synonym lists.
- Sensitive content in indexers, data sources, skillsets, and vectorizers. Sensitive content refers to connection strings, descriptions, identities, keys, and user inputs. For example, skillsets have Azure AI services keys, and some skills accept user inputs, such as custom entities. In both cases, keys and user inputs are encrypted. Any references to external resources (such as Azure data sources or Azure OpenAI models) are also encrypted.

If you require CMK across your search service, [set an enforcement policy](#).

Although you can't add encryption to an existing object, once an object is configured for encryption, you can change all parts of its encryption definition, including switching to a different key vault or HMS storage as long as the resource is in the same tenant.

Prerequisites

- [Azure AI Search](#) on a [billable tier](#) (Basic or higher, in any region).
- [Azure Key Vault](#) and a key vault with [soft-delete](#) and [purge protection](#) enabled. Or, [Azure Key Vault Managed HSM](#). This resource can be in any subscription, but it must be in the same tenant as Azure AI Search.
- Ability to set up permissions for key access and to assign roles. To create keys, you must be [Key Vault Crypto Officer](#) in Azure Key Vault or [Managed HSM Crypto Officer](#) in Azure Key Vault Managed HSM.

To assign roles, you must be subscription [Owner](#), [User Access Administrator](#), [Role-based Access Control Administrator](#), or be assigned to a custom role with [Microsoft.Authorization/roleAssignments/write](#) permissions.

Step 1: Create an encryption key

Use either Azure Key Vault or Azure Key Vault Managed HSM to create a key. Azure AI Search encryption supports RSA keys of sizes 2048, 3072 and 4096. For more information about supported key types, see [About keys](#).

We recommend reviewing [these tips](#) before you start.

Required operations are [Wrap](#), [Unwrap](#), [Encrypt](#), and [Decrypt](#).

You can [create a key vault using the Azure portal](#), [Azure CLI](#), or [Azure PowerShell](#).

1. Sign in to the [Azure portal](#) and open your key vault overview page.
2. Select **Objects > Keys** on the left, and then select **Generate/Import**.
3. In the **Create a key** pane, from the list of **Options**, choose **Generate** to create a new key.
4. Enter a **Name** for your key, and accept the defaults for other key properties.
5. Optionally, set a key rotation policy to [enable auto rotation](#).
6. Select **Create** to start the deployment.
7. After the key is created, get its key identifier. Select the key, select the current version, and then copy the key identifier. It's composed of the **key value Uri**, the **key name**, and the **key version**. You need the identifier to define an encrypted index in Azure AI Search. Recall that required operations are **Wrap**, **Unwrap**, **Encrypt**, and **Decrypt**.

[Home >](#)**aaaaaaaa-0b0b-1c1c-2d2d-333333333333**

...

Key Version



Discard changes



Download public key

Properties

Key type RSA

RSA key size 2048

Created 10/1/2024, 4:35:20 PM

Updated 10/1/2024, 4:35:20 PM

Key Identifier

[https://keyvault.vault.azure.net/keys/...](https://keyvault.vault.azure.net/keys/)

Settings

Set activation date Set expiration date Enabled

Tags 0 tags

Permitted operations

 Encrypt Decrypt Sign Verify Wrap Key Unwrap Key

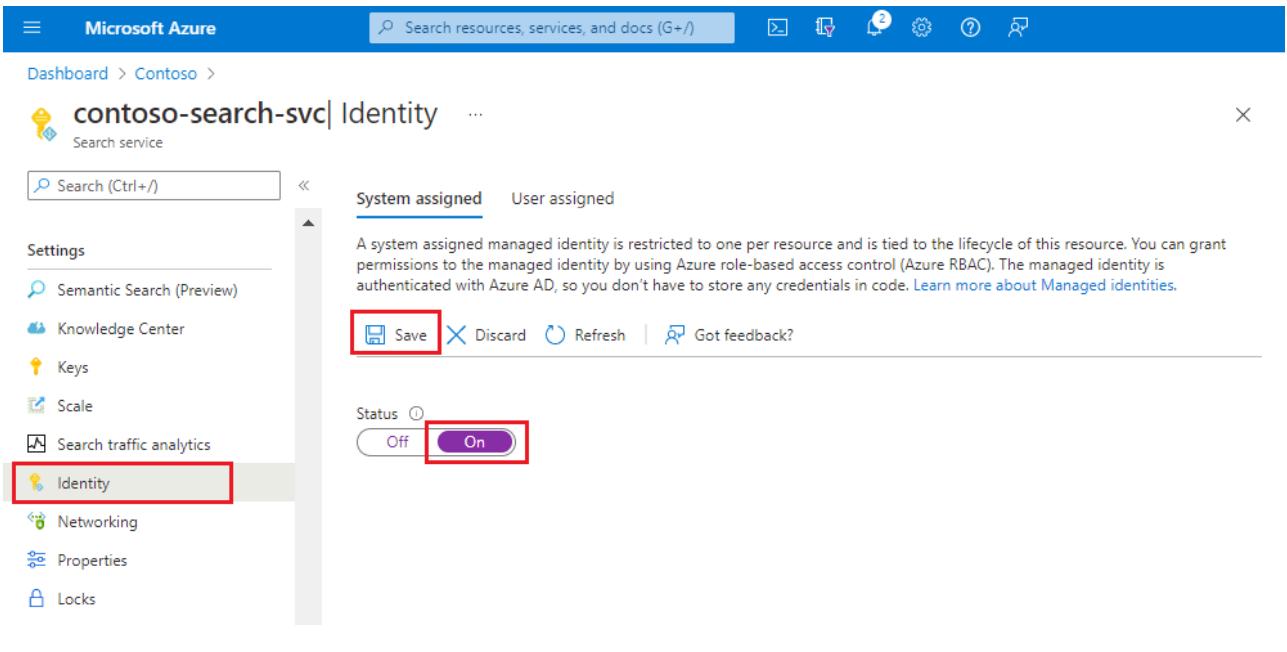
Step 2: Create a security principal

Create a security principal that your search service uses to access to the encryption key. You can use a managed identity and role assignment, or you can register an application and have the search service provide the application ID on requests.

We recommend using a managed identity and roles. You can use either a system-managed identity or user-managed identity. A managed identity enables your search service to authenticate through Microsoft Entra ID, without storing credentials (ApplicationID or ApplicationSecret) in code. The lifecycle of this type of managed identity is tied to the lifecycle of your search service, which can only have one system assigned managed identity. For more information about how managed identities work, see [What are managed identities for Azure resources](#).

System-managed identity

Enable the system-assigned managed identity for your search service. It's a two-click operation: enable and save.



The screenshot shows the Azure portal interface for managing a search service. The top navigation bar includes 'Microsoft Azure', a search bar, and various icons. Below the navigation is a breadcrumb trail: 'Dashboard > Contoso > contoso-search-svc Identity'. The main content area has a title 'contoso-search-svc| Identity' and tabs for 'System assigned' and 'User assigned'. A note explains that a system assigned managed identity is restricted to one per resource and is tied to the lifecycle of the resource. It mentions Azure RBAC and Azure AD authentication. At the bottom are buttons for 'Save' (highlighted), 'Discard', 'Refresh', and 'Got feedback?'. On the left, a sidebar lists 'Settings' (Semantic Search, Knowledge Center, Keys, Scale, Search traffic analytics), 'Networking', 'Properties', and 'Locks'. The 'Identity' option is specifically highlighted with a red box. The 'Status' section shows a toggle switch between 'Off' (gray) and 'On' (purple, highlighted with a red box). The overall theme is light blue and white.

Step 3: Grant permissions

If you configured your search service to use a managed identity, assign roles that give it access to the encryption key.

Role-based access control is recommended over the Access Policy permission model. For more information or migration steps, start with [Azure role-based access control \(Azure RBAC\) vs. access policies \(legacy\)](#).

1. Sign in to the [Azure portal](#) and find your key vault.
2. Select **Access control (IAM)** and select **Add role assignment**.
3. Select a role:
 - On Azure Key Vault, select **Key Vault Crypto Service Encryption User**.

- On Managed HSM, select **Managed HSM Crypto Service Encryption User**.
4. Select managed identities, select members, and then select the managed identity of your search service. If you're testing locally, assign this role to yourself as well.
5. Select **Review + Assign**.

Wait a few minutes for the role assignment to become operational.

Step 4: Encrypt content

Encryption occurs when you create or update an object. You can use the Azure portal for selected objects. For any object, use the [Search REST API](#) or an Azure SDK. Review the [Python example](#) in this article to see how content is encrypted programmatically.

Azure portal

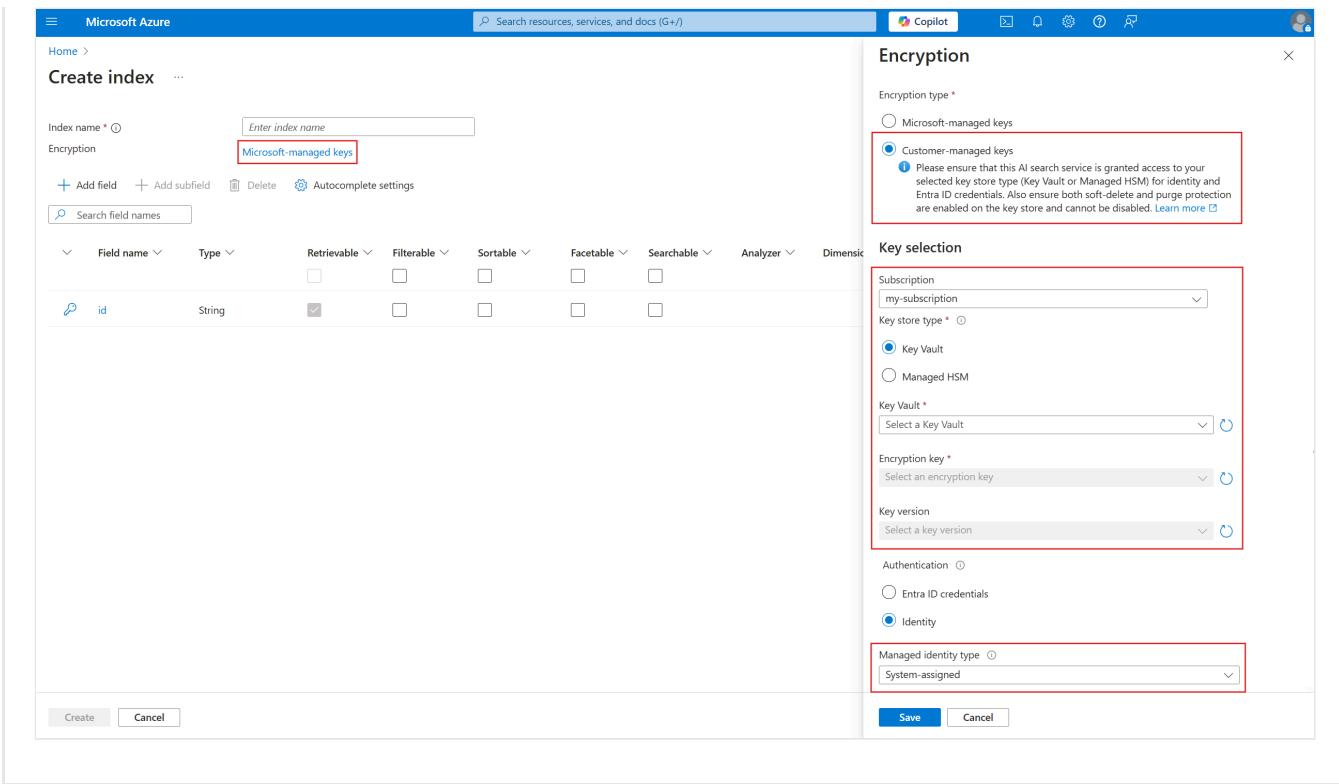
When you create a new object in the Azure portal, you can specify a predefined customer-managed key in a key vault. The Azure portal lets you enable CMK encryption for:

- Indexes
- Data sources
- Indexers

Requirements for using the Azure portal are that the key vault and key must exist, and you completed the previous steps for authorized access to the key.

In the Azure portal, skillsets are defined in JSON view. Use the JSON shown in the REST API examples to provide a customer-managed key on a skillset.

1. Sign in to the [Azure portal](#) and open your search service page.
2. Under **Search management**, select **Indexes**, **Indexers**, or **Data Sources**.
3. Add a new object. In the object definition, select **Microsoft-managed encryption**.
4. Select **Customer-managed keys** and choose your subscription, vault, key, and version.



Step 5: Test encryption

To verify encryption is working, revoke the encryption key, query the index (it should be unusable), and then reinstate the encryption key.

Use the Azure portal for this task. Make sure you have a role assignment that grants read access to the key.

1. On the Azure Key Vault page, select **Objects > Keys**.
2. Select the key you created, and then select **Delete**.
3. On the Azure AI Search page, select **Search management > Indexes**.
4. Select your index and use Search Explorer to run a query. You should get an error.
5. Return to the Azure Key Vault **Objects > Keys** page.
6. Select **Manage deleted keys**.
7. Select your key, and then select **Recover**.
8. Return to your index in Azure AI Search and rerun the query. You should see search results. If you don't see immediate results, wait a minute and try again.

Set up a policy to enforce CMK compliance

Azure policies help to enforce organizational standards and to assess compliance at-scale. Azure AI Search has two optional built-in policies related to CMK. These policies apply to new and existing search services.

[] Expand table

Effect	Effect if enabled
AuditIfNotExists	Checks for policy compliance: do objects have a customer-managed key defined, and is the content encrypted. This effect applies to existing services with content. It's evaluated each time an object is created or updated, or per the evaluation schedule . Learn more... ↗
Deny	Checks for policy enforcement: does the search service have SearchEncryptionWithCmk set to <code>Enabled</code> . This effect applies to new services only, which must be created with encryption enabled. Existing services remain operational but you can't update them unless you patch the service. None of the tools used for provisioning services expose this property, so be aware that setting the policy limits you to programmatic set up .

Assign a policy

1. In the Azure portal, navigate to a built-in policy and then select **Assign**.

- [AuditIfExists](#) ↗
- [Deny](#) ↗

Here's an example of the **AuditIfExists** policy in the Azure portal:



2. Set **policy scope** by selecting the subscription and resource group. Exclude any search services for which the policy shouldn't apply.
3. Accept or modify the defaults. Select **Review +create**, followed by **Create**.

Enable CMK policy enforcement

A policy that's assigned to a resource group in your subscription is effective immediately. Audit policies flag non-compliant resources, but Deny policies prevent the creation and update of non-compliant search services. This section explains how to create a compliant search service

or update a service to make it compliant. To bring objects into compliance, start at [step one](#) of this article.

Create a compliant search service

For new search services, create them with [SearchEncryptionWithCmk](#) set to `Enabled`.

Neither the Azure portal nor the command line tools (the Azure CLI and Azure PowerShell) provide this property natively, but you can use [Management REST API](#) to provision a search service with a CMK policy definition.

Management REST API

This example is from [Manage your Azure AI Search service with REST APIs](#), modified to include the [SearchEncryptionWithCmk](#) property.

```
rest

### Create a search service (provide an existing resource group)
@Resource-group = my-rg
@search-service-name = my-search
PUT
https://management.azure.com/subscriptions/{{subscriptionId}}/resourceGroups/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-service-name}}?api-version=2025-05-01 HTTP/1.1
Content-type: application/json
Authorization: Bearer {{token}}


{
    "location": "North Central US",
    "sku": {
        "name": "basic"
    },
    "properties": {
        "replicaCount": 1,
        "partitionCount": 1,
        "hostingMode": "default",
        "encryptionWithCmk": {
            "enforcement": "Enabled"
        }
    }
}
```

Update an existing search service

For existing search services that are now non-compliant, patch them using [Services - Update API](#) or the Azure CLI `az resource update` command. Patching the services restores the ability to update search service properties.

Management REST API

HTTP

```
PATCH https://management.azure.com/subscriptions/<your-subscription-Id>/resourceGroups/<your-resource-group-name>/providers/Microsoft.Search/searchServices/<your-search-service-name>?api-version=2025-05-01

{
  "properties": {
    "encryptionWithCmk": {
      "enforcement": "Enabled"
    }
  }
}
```

Rotate or update encryption keys

Use the following instructions to rotate keys or to migrate from Azure Key Vault to the Hardware Security Model (HSM).

For key rotation, we recommend using the [autorotation capabilities of Azure Key Vault](#). If you use autorotation, omit the key version in object definitions. The latest key is used, rather than a specific version.

When you change a key or its version, any object that uses the key must first be updated to use the new values **before** you delete the old values. Otherwise, the object becomes unusable because it can't be decrypted.

Recall that keys are cached for 60 minutes. Remember this when testing and rotating keys.

1. [Determine the key used by an index or synonym map.](#)
2. [Create a new key in key vault](#), but leave the original key available. In this step, you can switch from key vault to HSM.
3. [Update the encryptionKey properties](#) on an index or synonym map to use the new values. Only objects that were originally created with this property can be updated to use a different value.

4. Disable or delete the previous key in the key vault. Monitor key access to verify the new key is being used.

For performance reasons, the search service caches the key for up to several hours. If you disable or delete the key without providing a new one, queries continue to work on a temporary basis until the cache expires. However, once the search service can no longer decrypt content, you get this message: "Access forbidden. The query key used might have been revoked - please retry."

Key Vault tips

- If you're new to Azure Key Vault, review this quickstart to learn about basic tasks: [Set and retrieve a secret from Azure Key Vault using PowerShell](#).
- Use as many key vaults as you need. Managed keys can be in different key vaults. A search service can have multiple encrypted objects, each one encrypted with a different customer-managed encryption key, stored in different key vaults.
- Use the same [Azure tenant](#) so that you can retrieve your managed key through role assignments and by connecting through a system or user-managed identity. For more information about creating a tenant, see [Set up a new tenant](#).
- [Enable purge protection](#) and [soft-delete](#) on a key vault. Due to the nature of encryption with customer-managed keys, no one can retrieve your data if your Azure Key Vault key is deleted. To prevent data loss caused by accidental Key Vault key deletions, soft-delete and purge protection must be enabled on the key vault. Soft-delete is enabled by default, so you'll only encounter issues if you purposely disable it. Purge protection isn't enabled by default, but it's required for CMK encryption in Azure AI Search.
- [Enable logging](#) on the key vault so that you can monitor key usage.
- [Enable autorotation of keys](#) or follow strict procedures during routine rotation of key vault keys and application secrets and registration. Always update all [encrypted content](#) to use new secrets and keys before deleting the old ones. If you miss this step, your content can't be decrypted.

Work with encrypted content

With CMK encryption, you might notice latency for both indexing and queries due to the extra encrypt/decrypt work. Azure AI Search doesn't log encryption activity, but you can monitor key access through key vault logging.

We recommend that you [enable logging](#) as part of key vault configuration.

1. [Create a log analytics workspace](#).
2. [Add a diagnostic setting in key vault](#) that uses the workspace for data retention.
3. Select **audit** or **allLogs** for the category, give the diagnostic setting a name, and then save it.

Python example of an encryption key configuration

This section shows the Python representation of an `encryptionKey` in an object definition. The same definition applies to indexes, data sources, skillets, indexers, and synonym maps. To try this example on your search service and key vault, download the notebook from [azure-search-python-samples](#) ↗.

Install some packages.

Python

```
! pip install python-dotenv
! pip install azure-core
! pip install azure-search-documents==11.5.1
! pip install azure-identity
```

Create an index that has an encryption key.

Python

```
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import (
    SimpleField,
    SearchFieldDataType,
    SearchableField,
    SearchIndex,
    SearchResourceEncryptionKey
)
from azure.identity import DefaultAzureCredential

endpoint=<PUT YOUR AZURE SEARCH SERVICE ENDPOINT HERE>
credential = DefaultAzureCredential()

index_name = "test-cmk-index"
index_client = SearchIndexClient(endpoint=endpoint, credential=credential)
fields = [
    SimpleField(name="Id", type=SearchFieldDataType.String, key=True),
```

```

        SearchableField(name="Description", type=SearchFieldDataType.String)
    ]

scoring_profiles = []
suggester = []
encryption_key = SearchResourceEncryptionKey(
    key_name=<PUT YOUR KEY VAULT NAME HERE>,
    key_version=<PUT YOUR ALPHANUMERIC KEY VERSION HERE>,
    vault_uri=<PUT YOUR KEY VAULT ENDPOINT HERE>
)

index = SearchIndex(name=index_name, fields=fields, encryption_key=encryption_key)
result = index_client.create_or_update_index(index)
print(f' {result.name} created')

```

Get the index definition to verify encryption key configuration exists.

Python

```

index_name = "test-cmk-index-qs"
index_client = SearchIndexClient(endpoint=AZURE_SEARCH_SERVICE,
credential=credential)

result = index_client.get_index(index_name)
print(f"{result}")

```

Load the index with a few documents. All field content is considered sensitive and is encrypted on disk using your customer managed key.

Python

```

from azure.search.documents import SearchClient

# Create a documents payload
documents = [
    {
        "@search.action": "upload",
        "Id": "1",
        "Description": "The hotel is ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Time's Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities."
    },
    {
        "@search.action": "upload",
        "Id": "2",
        "Description": "The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts."
    },
]

```

```

    },
    "@search.action": "upload",
    "Id": "3",
    "Description": "The hotel stands out for its gastronomic excellence under the
management of William Dough, who advises on and oversees all of the Hotel's
restaurant services."
},
{
    "@search.action": "upload",
    "Id": "4",
    "Description": "The hotel is located in the heart of the historic center of
Sublime in an extremely vibrant and lively area within short walking distance to
the sites and landmarks of the city and is surrounded by the extraordinary beauty
of churches, buildings, shops and monuments. Sublime Palace is part of a lovingly
restored 1800 palace."
}
]

search_client = SearchClient(endpoint=AZURE_SEARCH_SERVICE, index_name=index_name,
credential=credential)
try:
    result = search_client.upload_documents(documents=documents)
    print("Upload of new document succeeded: {}".format(result[0].succeeded))
except Exception as ex:
    print(ex.message)

index_client = SearchClient(endpoint=AZURE_SEARCH_SERVICE,
credential=credential)

```

Run a query to confirm the index is operational.

Python

```

from azure.search.documents import SearchClient

query = "historic"

search_client = SearchClient(endpoint=AZURE_SEARCH_SERVICE, credential=credential,
index_name=index_name)

results = search_client.search(
    query_type='simple',
    search_text=query,
    select=["Id", "Description"],
    include_total_count=True
)

for result in results:
    print(f"Score: {result['@search.score']}")
    print(f"Id: {result['Id']}")
    print(f"Description: {result['Description']}")

```

Output from the query should produce results similar to the following example.

```
Score: 0.6130029
```

```
Id: 4
```

```
Description: The hotel is located in the heart of the historic center of Sublime in an extremely vibrant and lively area within short walking distance to the sites and landmarks of the city and is surrounded by the extraordinary beauty of churches, buildings, shops and monuments. Sublime Palace is part of a lovingly restored 1800 palace.
```

```
Score: 0.26286605
```

```
Id: 1
```

```
Description: The hotel is ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Time's Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.
```

Since encrypted content is decrypted prior to data refresh or queries, you won't see visual evidence of encryption. To verify encryption is working, check the resource logs.

Next steps

If you're unfamiliar with Azure security architecture, review the [Azure Security documentation](#), and in particular, this article:

[Data encryption-at-rest](#)

Configure customer-managed keys across different tenants

10/14/2025

When Azure Key Vault and Azure AI Search are in different Azure tenants, use a Microsoft Entra multitenant app to enable [customer-managed key \(CMK\) encryption](#) using a key from another tenant.

Prerequisites

- A tenant containing the search service that has content you want to encrypt. Azure AI Search must be [configured for role-based access](#). Support for CMK requires Basic pricing tier or higher.
- A separate tenant having the Azure Key Vault and the encryption keys you want to use. Azure Key Vault must be [configured for role-based access](#).
- Azure CLI for sending requests.

Create a multitenant Microsoft Entra application in tenant A

Use the Azure CLI to send requests. We refer to the tenant containing Azure AI Search as *tenant A*.

1. Get the tenant ID:

```
az account show --query tenantId --output tsv
```

2. Make sure you're signed in to tenant A:

```
az login --tenant <tenant-A-id>
```

3. Create the application registration:

```
az ad app create --display-name cross-tenant-auth --sign-in-audience
AzureADMultipleOrgs
```

4. Save the app ID output from this step.

Add a client secret to the multitenant application

1. To add the client secret to the multitenant application in tenant A, run the following command:

```
az ad app credential reset --id <multitenant-app-id>
```

2. Save the password output from this step. The password output is a required input for [setting up CMK](#) in Azure AI Search.

3. To specify when the client secret expires, you can specify an end-date parameter to this command.

```
az ad app credential reset --id <multitenant-app-id> --end-date <end-date>
```

The end-date parameter accepts a date in ISO 8601 format. For example: `az ad app credential reset --id <multitenant-app-id> --end-date 2026-12-31`.

Create a service principal in tenant B for the multitenant application

We refer to the tenant containing Azure Key Vault as *tenant B*. In tenant B, create a service principal for the multitenant application in tenant A.

1. Sign in to tenant B:

```
az login --tenant <tenant-B-id>
```

2. Create the service principal using the multitenant app ID output from the first step:

```
az ad sp create --id <multitenant-app-id>
```

This service principal is an instance of the multitenant application in tenant A. Roles assigned to this service principal in tenant B are also assigned to the multitenant application in tenant A.

3. Verify the link between tenant A and B by reviewing the "appOwnerOrganizationId" in the following command:

```
az ad sp show --id <multitenant-app-id>
```

This command displays the service principal details in JSON. Look for the "appOwnerOrganizationId" field in the output to confirm it matches tenant A's ID.

4. Save the object ID of the service principal (from the "id" field) from this step. The object ID is a required input for setting up CMK in Azure AI Search.

5. Get the resource ID for Azure Key Vault:

```
az keyvault show --name <key-vault-name> --query id --output tsv
```

6. Assign the **Key Vault Crypto Service Encryption User** role on the key vault in tenant B to the new service principal.

```
az role assignment create --assignee <service-principal-object-id> --role "Key Vault  
Crypto Service Encryption User" --scope <key-vault-resource-id>
```

An example of this assignment might look like this:

```
az role assignment create --assignee 12345678-1234-1234-1234-123456789012 --role  
"Key Vault Crypto Service Encryption User" --scope /subscriptions/87654321-4321-  
4321-4321-  
210987654321/resourceGroups/myKeyVaultRG/providers/Microsoft.KeyVault/vaults/myCompa  
nyKeyVault
```

Test encryption

Create a test index in the search service (tenant A) to validate the setup. Use the multitenant app ID and the credentials you added in the "access credentials" object to authenticate to the key vault in the other tenant.

You can use this sample index schema for testing. You can use the Azure portal to add an index and provide this JSON, or use a [REST client](#) to send a Create Index request.

JSON

```
{  
  "name": "cross-tenant-cmk-test",  
  "fields": [  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "key": true  
    }  
  ],  
  "encryptionKey": {  
    "keyVaultUri": "https://myCompanyKeyVault.vault.azure.net/",  
    "keyVaultKeyName": "search-encryption-key",  
    "keyVaultKeyVersion": "abc123def456ghi789",  
    "accessCredentials": {  
      "tenantId": "87654321-4321-4321-4321-210987654321",  
      "clientSecret": "MyClientSecret",  
      "objectType": "ServicePrincipal"  
    }  
  }  
}
```

```
        "applicationId": "12345678-1234-1234-1234-123456789012",
        "applicationSecret": "secretValueFromStep2"
    }
}
```

Verify the index was created successfully:

HTTP

```
GET https://<search-service>.search.windows.net/indexes/cross-tenant-cmk-test?api-version=2025-09-01
```

For more information about how to rotate or manage keys, see [Configure customer-managed keys for data encryption](#).

Find encrypted objects and information

Article • 04/14/2025

In Azure AI Search, customer-managed encryption keys are created, stored, and managed in Azure Key Vault. If you need to determine whether an object is encrypted, or what key name or version is used in Azure Key Vault, use the REST API or an Azure SDK to retrieve the **encryptionKey** property from the object definition in your search service.

Objects that aren't encrypted with a customer-managed key have an empty **encryptionKey** property. Otherwise, you might see a definition similar to the following example.

JSON

```
"encryptionKey":{  
    "keyVaultUri":"https://demokeyvault.vault.azure.net",  
    "keyVaultKeyName":"myEncryptionKey",  
    "keyVaultKeyVersion":"eaab6a663d59439ebb95ce2fe7d5f660",  
    "accessCredentials":{  
        "applicationId":"00001111-aaaa-2222-bbbb-3333cccc4444",  
        "applicationSecret":"myApplicationSecret"  
    }  
}
```

The **encryptionKey** construct is the same for all encrypted objects. It's a first-level property, on the same level as the object name and description.

Permissions for retrieving object definitions

You must have [Search Service Contributor](#) or equivalent permissions. To use [key-based authentication](#) instead, provide an admin API key. Admin permissions are required on requests that return object definitions and metadata. The easiest way to get the admin API key is through the Azure portal.

1. Sign in to the [Azure portal](#) and open the search service overview page.
2. On the left side, select **Keys** and copy an admin API.

For the remaining steps, switch to PowerShell and the REST API. The Azure portal doesn't show encryption key information for any object.

Retrieve object properties

Use PowerShell and REST to run the following commands to set up the variables and get object definitions.

Alternatively, you can also use the Azure SDK for [.NET](#), [Python](#), [JavaScript](#), and [Java](#).

First, connect to your Azure account.

```
PowerShell
```

```
Connect-AzAccount
```

If you have more than one active subscription in your tenant, specify the subscription containing your search service:

```
PowerShell
```

```
Set-AzContext -Subscription <your-subscription-ID>
```

Set up the headers used on each request in the current session. Provide the admin API key used for search service authentication.

```
PowerShell
```

```
$headers = @{
    'api-key' = '<YOUR-ADMIN-API-KEY>'
    'Content-Type' = 'application/json'
    'Accept' = 'application/json' }
```

To return a list of all search indexes, set the endpoint to the indexes collection.

```
PowerShell
```

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes?api-version=2024-07-01&$select=name'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

To return a specific index definition, provide its name in the path. The encryptionKey property is at the end.

```
PowerShell
```

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/<YOUR-INDEX-NAME>?api-version=2024-07-01'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

To return synonym maps, set the endpoint to the synonyms collection and then send the request.

```
PowerShell
```

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms?api-version=2024-07-01&$select=name'  
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

The following example returns a specific synonym map definition, including the encryptionKey property is towards the end of the definition.

```
PowerShell
```

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms/<YOUR-SYNONYM-MAP-NAME>?api-version=2024-07-01'  
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

Use the same pattern to return the encryptionKey property for other top-level objects such as indexers, skillsets, data sources, and index aliases.

Next steps

We recommend that you [enable logging](#) on Azure Key Vault so that you can monitor key usage.

For more information about using Azure Key or configuring customer managed encryption:

- [Quickstart: Set and retrieve a secret from Azure Key Vault using PowerShell](#)
- [Configure customer-managed keys for data encryption in Azure AI Search](#)

API versions in Azure AI Search

Azure AI Search rolls out feature updates regularly. Sometimes, but not always, these updates require a new version of the API to preserve backward compatibility. Publishing a new version allows you to control when and how you integrate search service updates in your code.

As a rule, the REST APIs and libraries are versioned only when necessary, since it can involve some effort to upgrade your code to use a new API version. A new version is needed only if some aspect of the API has changed in a way that breaks backward compatibility. Such changes can happen because of fixes to existing features, or because of new features that change existing API surface area.

For more information about the deprecation path, see the [Azure SDK lifecycle and support policy](#).

Deprecated versions

2023-07-01-preview was deprecated on April 8, 2024 and won't be supported after July 8, 2024.

This was the first REST API that offered vector search support. Newer API versions have a different vector configuration. You should [migrate to a newer version](#) as soon as possible.

Discontinued versions

Some API versions are discontinued and are no longer documented or supported:

- 2015-02-28
- 2015-02-28-Preview
- 2014-07-31-Preview
- 2014-10-20-Preview

All SDKs are based on REST API versions. If a REST version is discontinued, SDK packages based on that version are also discontinued. All Azure AI Search .NET SDKs older than [3.0.0-rc](#) are now obsolete.

Support for the above-listed versions ended on October 15, 2020. If you have code that uses a discontinued version, you can [migrate existing code](#) to a newer [REST API version](#) or to a newer Azure SDK.

REST APIs

[Expand table](#)

REST API	Link
Search Service (data plane)	See API versions in the REST API reference.
Search Management (control plane)	See API versions in the REST API reference.

Azure SDK for .NET

The following table provides links to more recent SDK versions.

[Expand table](#)

SDK version	Status	Change log	Description
Azure.Search.Documents 11	Active	Change Log ↗	APIs for data plane operations on a service, such as read-write operations on content and objects.
Azure.ResourceManager.Search ↗	Active	Change Log ↗	APIs for control plane operations on the search service.

Azure SDK for Java

[Expand table](#)

SDK version	Status	Change log	Description
azure-search-documents 11	Active	Change Log ↗	Use the <code>azure-search-documents</code> client library for data plane operations.
azure-resourcemanager-search 2	Active	Change Log ↗	Use the <code>azure-resourcemanager-search</code> client library for control plane operations.

Azure SDK for JavaScript

[Expand table](#)

SDK version	Status	Change log	Description
@azure/search-documents 12	Active	Change Log ↗	Use the <code>@azure/search-documents</code> client library for data plane operations.
@azure/arm-search 4	Active	Change Log ↗	Use the <code>@azure/arm-search</code> package for control plane operations.

Azure SDK for Python

[Expand table](#)

SDK version	Status	Change log	Description
azure-search-documents 11	Active	Change Log ↗	Use the <code>azure-search-documents</code> client library for data plane operations.
azure-mgmt-search 9 ↗	Active	Change Log ↗	Use the <code>azure-mgmt-search</code> client library for control plane operations.

All Azure SDKs

If you're looking for beta client libraries and documentation, [this page ↗](#) contains links to all of the Azure SDK library packages, code, and docs.

Last updated on 11/21/2025

Preview features in Azure AI Search

This article identifies all data plane and control plane features in public preview. This list is helpful for checking feature status. It also explains how to call a preview REST API.

Preview API versions are cumulative and roll up to the next preview. We recommend always using the latest preview APIs for full access to all preview features.

Preview features are removed from this list if they're retired or transition to general availability. For announcements regarding general availability and retirement, see [Service Updates](#) or [What's New](#).

Data plane preview features

[+] [Expand table](#)

Feature	Description	Availability
Agentic retrieval	<p>Create a conversational search experience powered by large language models (LLMs). Agentic retrieval breaks down complex user queries into subqueries, runs the subqueries simultaneously, reranks the results for relevance, and either extracts grounding data or synthesizes answers into natural language. To get started, see Quickstart: Agentic retrieval.</p> <p>The pipeline involves one or more knowledge sources within a knowledge base, whose response payload provides full transparency into the query plan and reference data.</p> <p>Knowledge sources and knowledge bases created in the Azure portal use the 2025-08-01-preview schema and aren't compatible with the latest 2025-11-01-preview. To get started, see Quickstart: Agentic retrieval in the Azure portal.</p>	Knowledge Sources (preview) , Knowledge bases (preview) , Knowledge Retrieval (preview) , and the Azure portal
Purview index configuration	Apply Microsoft Purview classifications and sensitivity labels to indexed content based on source metadata for enhanced data governance.	Create or Update Index (preview)
Scoring function aggregation	Combine and aggregate multiple scoring functions, enabling more sophisticated	Create or Update Index (preview)

Feature	Description	Availability
	relevance customization and weighted signal combination.	
Facet aggregations	Use sum, count, minimum, maximum, and other aggregate functions to provide enhanced analytics in faceted search experiences.	Search Documents (preview)
Improved indexer runtime tracking information	Cumulative indexer processing information for the search service and for specific indexers.	Get Service Statistics (preview) and Get Status - Indexers (preview)
Strict postfiltering for vector queries	Adds the <code>strictPostFilter</code> mode to the <code>vectorFilterMode</code> parameter. When specified, filters are applied after the global top- k vector results are identified, ensuring that returned documents are a subset of the unfiltered results.	Search Documents (preview)
Multivector support	Index multiple child vectors within a single document field. You can now use vector types in nested fields of complex collections, effectively allowing multiple vectors to be associated with a single document.	Create or Update Index (preview)
Document-level access control	Flow document-level permissions from blobs in Azure Data Lake Storage (ADLS) Gen2 to searchable documents in an index. Queries can now filter results based on user identity for selected data sources.	Create or Update Index (preview)
GenAI Prompt skill	Skill that connects to a large language model (LLM) for information using a prompt you provide. With this skill, you can populate a searchable field using content from an LLM. A primary use case for this skill is <i>image verbalization</i> , using an LLM to describe images and send the description to a searchable field in your index.	Create or Update Skillset (preview)
flightingOptIn parameter in a semantic configuration	Opt in to use prerelease semantic ranking models if one is available in a search service region.	Create or Update Index (preview)
Facet hierarchies, aggregations, and facet filters	New facet query parameters support nested facets. For numeric facetable fields, you can sum the values of each field. You can also	Search Documents (preview)

Feature	Description	Availability
	specify filters on a facet to add inclusion or exclusion criteria.	
Query rewrite in the semantic reranker	Set options on a semantic query to rewrite the query input into a revised or expanded query that generates more relevant results from the L2 ranker.	Search Documents (preview)
Keyless billing for Azure AI skills processing.	Use a managed identity and roles for a keyless connection to Foundry Tools for built-in skills processing. This capability removes restrictions for having both search and Foundry Tools in the same region.	Create or Update Skillset (preview)
Markdown parsing mode	With this parsing mode, indexers can generate one-to-one or one-to-many search documents from Markdown files in Azure Storage.	Create or Update Indexer (preview)
Target filters in a hybrid search to just the vector queries	A filter on a hybrid query involves all subqueries on the request, regardless of type. You can override the global filter to scope the filter to a specific subquery. A new <code>filterOverride</code> parameter provides the behaviors.	Search Documents (preview)
Text Split skill (token chunking)	This skill has new parameters that improve data chunking for embedding models. A new <code>unit</code> parameter lets you specify token chunking. You can now chunk by token length, setting the length to a value that makes sense for your embedding model. You can also specify the tokenizer and any tokens that shouldn't be split during data chunking.	Create or Update Skillset (preview)
Azure Vision multimodal embedding skill	Skill that calls the Azure Vision multimodal API to generate embeddings for text or images during indexing.	Create or Update Skillset (preview)
Azure Machine Learning (AML) skill	AML skill integrates an inferencing endpoint from Azure Machine Learning. In previous preview APIs, it supports connections to deployed custom models in an AML workspace. Starting in the 2025-11-01-preview, you can use this skill in workflows that connect to embedding models in the Microsoft Foundry model catalog. It's also available in the Azure portal, in skillset design, assuming Azure AI Search and Azure Machine Learning services are deployed in the same subscription.	Create or Update Skillset (preview)

Feature	Description	Availability
Incremental enrichment cache	Adds caching to an enrichment pipeline, allowing you to reuse existing output if a targeted modification, such as an update to a skillset or another object, doesn't change the content. Caching applies only to enriched documents produced by a skillset.	Create or Update Indexer (preview)
Azure Files indexer	Data source for indexer-based indexing from Azure Files .	Create or Update Data Source (preview)
SharePoint indexer	Data source for indexer-based indexing of SharePoint content.	Sign up to enable the feature. Create or Update Data Source (preview) or the Azure portal.
MySQL indexer	Data source for indexer-based indexing of Azure MySQL data sources.	Sign up to enable the feature. Create or Update Data Source (preview) , .NET SDK 11.2.1, and Azure portal
Azure Cosmos DB for MongoDB indexer	Data source for indexer-based indexing through the MongoDB APIs in Azure Cosmos DB.	Sign up to enable the feature. Create or Update Data Source (preview) or the Azure portal.
Azure Cosmos DB for Apache Gremlin indexer	Data source for indexer-based indexing through the Apache Gremlin APIs in Azure Cosmos DB.	Sign up to enable the feature. Create or Update Data Source (preview) .
Native blob soft delete	Applies to the Azure Blob Storage indexer. Recognizes blobs that are in a soft-deleted state, and removes the corresponding search document during indexing.	Create or Update Data Source (preview) .
Reset Documents	Reprocesses individually selected search documents in indexer workloads.	Reset Documents (preview) .

Feature	Description	Availability
speller	Optional spelling correction on query term inputs for simple, full, and semantic queries.	Search Documents (preview)
featuresMode parameter	BM25 relevance score expansion to include details: per field similarity score, per field term frequency, and per field number of unique tokens matched. You can consume these data points in custom scoring solutions .	Search Documents (preview)
vectorQueries.threshold parameter	Exclude low-scoring search result based on a minimum score.	Search Documents (preview)
hybridSearch.maxTextRecallSize and countAndFacetMode parameters	Adjust the inputs to a hybrid query by controlling the amount BM25-ranked results that flow to the hybrid ranking model.	Search Documents (preview)
moreLikeThis	Finds documents that are relevant to a specific document. This feature has been in earlier previews.	Search Documents (preview)

Control plane preview features

Currently, there are no control plane features in preview.

Preview features in Azure SDKs

Each Azure SDK team releases beta packages on their own timeline. Check the change log for mentions of new features in beta packages:

- [Change log for Azure SDK for .NET](#)
- [Change log for Azure SDK for Java](#)
- [Change log for Azure SDK for JavaScript](#)
- [Change log for Azure SDK for Python](#).

Using preview features

Experimental features are available through the preview REST API first, followed by Azure portal, and then the Azure SDKs.

The following statements apply to preview features:

- Preview features are available under [Supplemental Terms of Use](#), without a service level agreement.
- Preview features might undergo breaking changes if a redesign is required.
- Sometimes preview features don't make it into a GA release.

If you write code against a preview API, you should prepare to upgrade that code to newer API versions when they roll out. We maintain an [Upgrade REST APIs](#) document to make that step easier.

How to call a preview REST API

Preview REST APIs are accessed through the api-version parameter on the URI. Older previews are still operational but become stale over time and aren't updated with new features or bug fixes.

For data plane operations on content, [2025-11-01-preview](#) is the most recent preview version. The following example shows the syntax for [Indexes GET \(preview\)](#):

```
rest
GET {endpoint}/indexes('{indexName}')?api-version=2025-11-01-Preview
```

For management operations on the search service, [2025-05-01-preview](#) is the most recent preview version. The following example shows the syntax for Update Service 2025-05-01-preview version.

```
rest
PATCH
https://management.azure.com/subscriptions/subid/resourceGroups/rg1/providers/Microsoft.Search/searchServices/mysearchservice?api-version=2025-05-01-preview

{
  "tags": {
    "app-name": "My e-commerce app",
    "new-tag": "Adding a new tag"
  },
  "properties": {
    "replicaCount": 2
  }
}
```

See also

- [Quickstart: Full-text search using REST APIs](#)
 - [Search REST API overview](#)
 - [Search REST API versions](#)
 - [Manage using the REST APIs](#)
 - [Management REST API overview](#)
 - [Management REST API versions](#)
-

Last updated on 11/18/2025

Upgrade to the latest REST API in Azure AI Search

Use this article to migrate to newer versions of the [Search Service REST APIs](#) and the [Search Management REST APIs](#) for [data plane and control plane](#) operations.

Here are the most recent versions of the REST APIs:

[+] [Expand table](#)

Targeted operations	REST API	Status
Data plane	2025-09-01	Stable
Data plane	2025-11-01-preview	Preview
Control plane	2025-05-01	Stable
Control plane	2025-02-01-preview	Preview

Upgrade instructions focus on code changes that get you through breaking changes from previous versions so that existing code runs the same as before, but on the newer API version. Once your code is in working order, you can decide whether to adopt newer features. To learn more about new features, see [What's New](#).

We recommend upgrading API versions in succession, working through each version until you get to the newest one.

[2023-07-01-preview](#) was the first REST API for vector support. **Do not use this API version.** It's now deprecated and you should migrate to either stable or newer preview REST APIs immediately.

(!) Note

REST API reference docs are now versioned. For version-specific content, open a reference page and then use the selector located above the table of contents, to pick your version.

When to upgrade

Azure AI Search breaks backward compatibility as a last resort. Upgrade is necessary when:

- Your code references a retired or unsupported API version and is subject to one or more breaking changes. You must address breaking changes if your code targets [2025-11-01-](#)

[preview](#) for agentic retrieval, [2025-05-01-preview](#) for knowledge agents, [2023-07-10-preview](#) for vectors, [2020-06-01-preview](#) for semantic ranker, and [2019-05-06](#) for obsolete skills and workarounds.

- Your code fails when unrecognized properties are returned in an API response. As a best practice, your application should ignore properties that it doesn't understand.
- Your code persists API requests and tries to resend them to the new API version. For example, this might happen if your application persists continuation tokens returned from the Search API (for more information, look for `@search.nextPageParameters` in the [Search API Reference](#)).

How to upgrade

1. If you're upgrading a data plane version, review the [release notes](#) for the new API version.
2. Update the `api-version` parameter, specified in the request header, to a newer version.

In your application code that makes direct calls to the REST APIs, search for all instances of the existing version and then replace it with the new version. For more information about structuring a REST call, see [Quickstart: Full-text search using REST](#).

If you're using an Azure SDK, those packages target specific versions of the REST API. Package updates might coincide with a REST API update, but each SDK is on its own release schedule that ships independently of Azure AI Search REST API versions. Check the change log of your SDK package to determine whether a package release targets the latest REST API version.

3. If you're upgrading a data plane version, review the breaking changes documented in this article and implement the workarounds. Start with the version used by your code and resolve any breaking change for each newer API version until you get to the newest stable or preview release.

Breaking changes

The following breaking changes apply to data operations.

Breaking changes for agentic retrieval

The latest `2025-11-01-preview` refactors the APIs for knowledge agents (bases), knowledge sources, and the retrieve action. The latest `2025-11-01-preview` renames knowledge agents to

knowledge bases and relocates several properties. Several properties are replaced or relocated to other objects.

For help with breaking changes, see [Migrate your agentic retrieval code](#).

Breaking changes for knowledge agents

[Knowledge agents](#) were introduced in `2025-05-01-preview`. In `2025-08-01-preview`, `targetIndexes` was replaced with a new knowledge source object and `defaultMaxDocsForReranker` was replaced with other APIs. More breaking changes are introduced in `2025-11-01-preview`.

For help with breaking changes, see [Migrate your agentic retrieval code](#).

Breaking changes for client code that reads connection information

Effective March 29, 2024 and applicable to all [supported REST APIs](#):

- [GET Skillset](#), [GET Index](#), and [GET Indexer](#) no longer return keys or connection properties in a response. This is a breaking change if you have downstream code that reads keys or connections (sensitive data) from a GET response.
- If you need to retrieve admin or query API keys for your search service, use the [Search Management REST APIs](#).
- If you need to retrieve connection strings of another Azure resource such as Azure Storage or Azure Cosmos DB, use the APIs of that resource and published guidance to obtain the information.

Breaking changes for semantic ranker

[Semantic ranker](#) became generally available in `2023-11-01`. These are the breaking changes from earlier releases:

- In all versions after `2020-06-01-preview`: `semanticConfiguration` replaces `searchFields` as the mechanism for specifying which fields to use for L2 ranking.
- For all API versions, updates on July 14, 2023 to the Microsoft-hosted semantic models made semantic ranker language-agnostic, effectively decommissioning the `queryLanguage` property. There's no "breaking change" in code, but the property is ignored.

See [Migrate from preview version](#) to transition your code to use `semanticConfiguration`.

Data plane upgrades

Upgrade guidance assumes upgrade from the most recent previous version. If your code is based on an old API version, we recommend upgrading through each successive version to get to the newest version.

Upgrade to 2025-11-01-preview

[2025-11-01-preview](#) introduces the following breaking changes to agentic retrieval as implemented in the [2025-08-01-preview](#):

- Replaces `agents` with `knowledgebases`. Several properties related to knowledge sources moved out of the knowledge base definition and to the retrieve action.
- Knowledge source properties are refactored, implementing a new `ingestionParameters` object for knowledge sources that generate an indexer pipeline.

For more information on changes and code migration, see [breaking changes in 2025-11-01-preview](#) and [How to migrate](#).

For all other existing APIs, there are no behavior changes. You can swap in the new API version and your code runs the same as before.

Upgrade to 2025-09-01

[2025-09-01](#) is the latest stable REST API version and it adds general availability for the OneLake indexer, Document Layout skill, and other APIs.

There are no breaking changes if you're upgrading from [2024-07-01](#) and not using any preview features. To use the new stable release, change the API version and test your code.

Upgrade to 2025-08-01-preview

[2025-08-01-preview](#) introduces the following breaking changes to knowledge agents created using [2025-05-01-preview](#):

- Replaces `targetIndexes` with `knowledgeSources`.
- Removes `defaultMaxDocsForReranker` without replacement.

Otherwise, there are no behavior changes on existing APIs. You can swap in the new API version and your code runs the same as before.

Upgrade to 2025-05-01-preview

[2025-05-01-preview](#) provides new features, but there are no behavior changes on existing APIs. You can swap in the new API version and your code runs the same as before.

Upgrade to 2025-03-01-preview

[2025-03-01-preview](#) provides new features, but there are no behavior changes on existing APIs. You can swap in the new API version and your code runs the same as before.

Upgrade to 2024-11-01-preview

[2024-11-01-preview](#) query rewrite, Document Layout skill, keyless billing for skills processing, Markdown parsing mode, and rescore options for compressed vectors.

If you're upgrading from [2024-09-01-preview](#), you can swap in the new API version and your code runs the same as before.

However, the new version introduces syntax changes to `vectorSearch.compressions`:

- Replaces `rerankWithOriginalVectors` with `enableRescoring`
- Moves `defaultOversampling` to a new `rescoringOptions` property object

Backwards compatibility is preserved due to an internal API mapping, but we recommend changing the syntax if you adopt the new preview version. For a comparison of the syntax, see [Compress vectors using scalar or binary quantization](#).

Upgrade to 2024-09-01-preview

[2024-09-01-preview](#) adds Matryoshka Representation Learning (MRL) compression for text-embedding-3 models, targeted vector filtering for hybrid queries, vector subscore details for debugging, and token chunking for [Text Split skill](#).

If you're upgrading from [2024-05-01-preview](#), you can swap in the new API version and your code runs the same as before.

Upgrade to 2024-07-01

[2024-07-01](#) is a general release. The former preview features are now generally available: integrated chunking and vectorization (Text Split skill, AzureOpenAIEmbedding skill), query vectorizer based on AzureOpenAIEmbedding, vector compression (scalar quantization, binary quantization, stored property, narrow data types).

There are no breaking changes if you upgrade from `2024-05-01-preview` to stable. To use the new stable release, change the API version and test your code.

There are breaking changes if you upgrade directly from `2023-11-01`. Follow the steps outlined for each newer preview to migrate from `2023-11-01` to `2024-07-01`.

Upgrade to 2024-05-01-preview

[2024-05-01-preview](#) adds an indexer for Microsoft OneLake, binary vectors, and more embedding models.

If you're upgrading from `2024-03-01-preview`, the AzureOpenAIEmbedding skill now requires a model name and dimensions property.

1. Search your codebase for [AzureOpenAIEmbedding](#) references.
2. Set `modelName` to "text-embedding-ada-002" and set `dimensions` to "1536".

Upgrade to 2024-03-01-preview

[2024-03-01-preview](#) adds narrow data types, scalar quantization, and vector storage options.

If you're upgrading from `2023-10-01-preview`, there are no breaking changes. However, there's one behavior difference: for `2023-11-01` and newer previews, the `vectorFilterMode` default changed from postfilter to prefilter for [filter expressions](#).

1. Search your codebase for `vectorFilterMode` references.
2. If the property is explicitly set, no action is required. If you relied on the default value, the new default behavior is to filter *before* query execution. If you want post-query filtering, explicitly set `vectorFilterMode` to postfilter to retain the old behavior.

Upgrade to 2023-11-01

[2023-11-01](#) is a general release. The former preview features are now generally available: semantic ranker and vector support.

There are no breaking changes from `2023-10-01-preview`, but there are multiple breaking changes from `2023-07-01-preview` to `2023-11-01`. For more information, see [Upgrade from 2023-07-01-preview](#).

To use the new stable release, change the API version and test your code.

Upgrade to 2023-10-01-preview

[2023-10-01-preview](#) was the first preview version to add [built-in data chunking and vectorization during indexing](#) and [built-in query vectorization](#). It also supports vector indexing and queries from the previous version.

If you're upgrading from the previous version, the next section has the steps.

Upgrade from 2023-07-01-preview

Don't use this API version. It implements a vector query syntax that's incompatible with any newer API version.

[2023-07-01-preview](#) is now deprecated, so you shouldn't base new code on this version, nor should you upgrade to this version under any circumstances. This section explains the migration path from [2023-07-01-preview](#) to any newer API version.

Portal upgrade for vector indexes

Azure portal supports a one-click upgrade path for [2023-07-01-preview](#) indexes. It detects vector fields and provides a **Migrate** button.

- Migration path is from [2023-07-01-preview](#) to [2024-05-01-preview](#).
- Updates are limited to vector field definitions and vector search algorithm configurations.
- Updates are one-way. You can't reverse the upgrade. Once the index is upgraded, you must use [2024-05-01-preview](#) or later to query the index.

There's no portal migration for upgrading vector query syntax. See [code upgrades](#) for query syntax changes.

Before selecting **Migrate**, select **Edit JSON** to review the updated schema first. You should find a schema that conforms to the changes described in the [code upgrade](#) section. Portal migration only handles indexes with one vector search algorithm configuration. It creates a default profile that maps to the [2023-07-01-preview](#) vector search algorithm. Indexes with multiple vector search configurations require manual migration.

Code upgrade for vector indexes and queries

[Vector search](#) support was introduced in [Create or Update Index \(2023-07-01-preview\)](#).

Upgrading from [2023-07-01-preview](#) to any newer stable or preview version requires:

- Renaming and restructuring the vector configuration in the index
- Rewriting your vector queries

Use the instructions in this section to migrate vector fields, configuration, and queries from `2023-07-01-preview`.

1. Call [Get Index](#) to retrieve the existing definition.
2. Modify the vector search configuration. `2023-11-01` and later versions introduce the concept of *vector profiles* that bundle vector-related configurations under one name. Newer versions also rename `algorithmConfigurations` to `algorithms`.
 - Rename `algorithmConfigurations` to `algorithms`. This is only a renaming of the array. The contents are backwards compatible. This means your existing HNSW configuration parameters can be used.
 - Add `profiles`, giving a name and an algorithm configuration for each one.

Before migration (2023-07-01-preview):

HTTP

```
"vectorSearch": {
  "algorithmConfigurations": [
    {
      "name": "myHnswConfig",
      "kind": "hnsw",
      "hnswParameters": {
        "m": 4,
        "efConstruction": 400,
        "efSearch": 500,
        "metric": "cosine"
      }
    }
  ]
}
```

After migration (2023-11-01):

HTTP

```
"vectorSearch": {
  "algorithms": [
    {
      "name": "myHnswConfig",
      "kind": "hnsw",
      "hnswParameters": {
        "m": 4,
        "efConstruction": 400,
        "efSearch": 500,
        "metric": "cosine"
      }
    }
  ]
}
```

```

        "metric": "cosine"
    }
}
],
"profiles": [
{
    "name": "myHnswProfile",
    "algorithm": "myHnswConfig"
}
]
}

```

3. Modify vector field definitions, replacing `vectorSearchConfiguration` with `vectorSearchProfile`. Make sure the profile name resolves to a new vector profile definition, and not the algorithm configuration name. Other vector field properties remain unchanged. For example, they can't be filterable, sortable, or facetable, nor use analyzers or normalizers or synonym maps.

Before (2023-07-01-preview):

HTTP

```
{
    "name": "contentVector",
    "type": "Collection(Edm.Single)",
    "key": false,
    "searchable": true,
    "retrievable": true,
    "filterable": false,
    "sortable": false,
    "facetable": false,
    "analyzer": "",
    "searchAnalyzer": "",
    "indexAnalyzer": "",
    "normalizer": "",
    "synonymMaps": "",
    "dimensions": 1536,
    "vectorSearchConfiguration": "myHnswConfig"
}
```

After (2023-11-01):

HTTP

```
{
    "name": "contentVector",
    "type": "Collection(Edm.Single)",
    "searchable": true,
    "retrievable": true,
    "filterable": false,
    "sortable": false,
```

```

    "facetable": false,
    "analyzer": "",
    "searchAnalyzer": "",
    "indexAnalyzer": "",
    "normalizer": "",
    "synonymMaps": "",
    "dimensions": 1536,
    "vectorSearchProfile": "myHnswProfile"
}

```

4. Call [Create or Update Index](#) to post the changes.

5. Modify [Search POST](#) to change the query syntax. This API change enables support for polymorphic vector query types.

- Rename `vectors` to `vectorQueries`.
- For each vector query, add `kind`, setting it to `vector`.
- For each vector query, rename `value` to `vector`.
- Optionally, add `vectorFilterMode` if you're using [filter expressions](#). The default is prefilter for indexes created after `2023-10-01`. Indexes created before that date only support postfilter, regardless of how you set the filter mode.

Before (2023-07-01-preview):

HTTP

```
{
  "search": (this parameter is ignored in vector search),
  "vectors": [
    {
      "value": [
        0.103,
        0.0712,
        0.0852,
        0.1547,
        0.1183
      ],
      "fields": "contentVector",
      "k": 5
    }
  ],
  "select": "title, content, category"
}
```

After (2023-11-01):

HTTP

```
{  
  "search": "(this parameter is ignored in vector search)",  
  "vectorQueries": [  
    {  
      "kind": "vector",  
      "vector": [  
        0.103,  
        0.0712,  
        0.0852,  
        0.1547,  
        0.1183  
      ],  
      "fields": "contentVector",  
      "k": 5  
    }  
  ],  
  "vectorFilterMode": "preFilter",  
  "select": "title, content, category"  
}
```

These steps complete the migration to `2023-11-01` stable API version or newer preview API versions.

Upgrade to 2020-06-30

In this version, there's one breaking change and several behavioral differences. Generally available features include:

- [Knowledge store](#), persistent storage of enriched content created through skillsets, created for downstream analysis and processing through other applications. A knowledge store is created through Azure AI Search REST APIs but it resides in Azure Storage.

Breaking change

Code written against earlier API versions breaks on `2020-06-30` and later if code contains the following functionality:

- Any `Edm.Date` literals (a date composed of year-month-day, such as `2020-12-12`) in filter expressions must follow the `Edm.DateTimeOffset` format: `2020-12-12T00:00:00Z`. This change was necessary to handle erroneous or unexpected query results due to timezone differences.

Behavior changes

- [BM25 ranking algorithm](#) replaces the previous ranking algorithm with newer technology. Services created after 2019 use this algorithm automatically. For older services, you must set parameters to use the new algorithm.
- Ordered results for null values have changed in this version, with null values appearing first if the sort is `asc` and last if the sort is `desc`. If you wrote code to handle how null values are sorted, be aware of this change.

Upgrade to 2019-05-06

Features that became generally available in this API version include:

- [Autocomplete](#) is a typeahead feature that completes a partially specified term input.
- [Complex types](#) provides native support for structured object data in search index.
- [JsonLines parsing modes](#), part of Azure Blob indexing, creates one search document per JSON entity that is separated by a newline.
- [AI enrichment](#) provides indexing that uses the AI enrichment engines of Foundry Tools.

Breaking changes

Code written against an earlier API version breaks on `2019-05-06` and later if it contains the following functionality:

1. Type property for Azure Cosmos DB. For indexers targeting an [Azure Cosmos DB for NoSQL API](#) data source, change `"type": "documentdb"` to `"type": "cosmosdb"`.
2. If your indexer error handling includes references to the `status` property, you should remove it. We removed status from the error response because it wasn't providing useful information.
3. Data source connection strings are no longer returned in the response. From API versions `2019-05-06` and `2019-05-06-Preview` onwards, the data source API no longer returns connection strings in the response of any REST operation. In previous API versions, for data sources created using POST, Azure AI Search returned `201` followed by the OData response, which contained the connection string in plain text.
4. Named Entity Recognition cognitive skill is retired. If you called the [Name Entity Recognition](#) skill in your code, the call fails. Replacement functionality is [Entity Recognition Skill \(V3\)](#). Follow the recommendations in [Deprecated skills](#) to migrate to a supported skill.

Upgrading complex types

API version `2019-05-06` added formal support for complex types. If your code implemented previous recommendations for complex type equivalency in 2017-11-11-Preview or 2016-09-01-Preview, there are some new and changed limits starting in version `2019-05-06` of which you need to be aware:

- The limits on the depth of subfields and the number of complex collections per index have been lowered. If you created indexes that exceed these limits using the preview api-versions, any attempt to update or recreate them using API version `2019-05-06` fails. If you find yourself in this situation, you need to redesign your schema to fit within the new limits and then rebuild your index.
- There's a new limit starting in api-version `2019-05-06` on the number of elements of complex collections per document. If you created indexes with documents that exceed these limits using the preview api-versions, any attempt to reindex that data using api-version `2019-05-06` fails. If you find yourself in this situation, you need to reduce the number of complex collection elements per document before reindexing your data.

For more information, see [Service limits for Azure AI Search](#).

How to upgrade an old complex type structure

If your code is using complex types with one of the older preview API versions, you might be using an index definition format that looks like this:

JSON

```
{  
  "name": "hotels",  
  "fields": [  
    { "name": "HotelId", "type": "Edm.String", "key": true, "filterable": true },  
    { "name": "HotelName", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": true, "facetable": false },  
    { "name": "Description", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false, "analyzer": "en.microsoft" },  
    { "name": "Description_fr", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false, "analyzer": "fr.microsoft" },  
    { "name": "Category", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true },  
    { "name": "Tags", "type": "Collection(Edm.String)", "searchable": true, "filterable": true, "sortable": false, "facetable": true, "analyzer": "tagsAnalyzer" },  
    { "name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true, "sortable": true, "facetable": true },  
    { "name": "LastRenovationDate", "type": "Edm.DateTimeOffset", "filterable": true, "sortable": true, "facetable": true },  
    { "name": "Rating", "type": "Edm.Double", "filterable": true, "sortable": true, "facetable": true }]
```

```

"facetable": true },
  { "name": "Address", "type": "Edm.ComplexType" },
    { "name": "Address/StreetAddress", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false, "searchable": true },
      { "name": "Address/City", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
        { "name": "Address/StateProvince", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
          { "name": "Address/PostalCode", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
            { "name": "Address/Country", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
              { "name": "Location", "type": "Edm.GeographyPoint", "filterable": true,
"sortable": true },
                { "name": "Rooms", "type": "Collection(Edm.ComplexType)" },
                  { "name": "Rooms/Description", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer": "en.lucene"
},
                    { "name": "Rooms/Description_fr", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer": "fr.lucene"
},
                      { "name": "Rooms/Type", "type": "Edm.String", "searchable": true },
                        { "name": "Rooms/BaseRate", "type": "Edm.Double", "filterable": true,
"facetable": true },
                          { "name": "Rooms/BedOptions", "type": "Edm.String", "searchable": true },
                            { "name": "Rooms/SleepsCount", "type": "Edm.Int32", "filterable": true,
"facetable": true },
                              { "name": "Rooms/SmokingAllowed", "type": "Edm.Boolean", "filterable": true,
"facetable": true },
                                { "name": "Rooms/Tags", "type": "Collection(Edm.String)", "searchable": true,
"filterable": true, "facetable": true, "analyzer": "tagsAnalyzer" }
      ]
}

```

A newer tree-like format for defining index fields was introduced in API version [2017-11-11-Preview](#). In the new format, each complex field has a `fields` collection where its subfields are defined. In API version 2019-05-06, this new format is used exclusively and attempting to create or update an index using the old format will fail. If you have indexes created using the old format, you'll need to use API version [2017-11-11-Preview](#) to update them to the new format before they can be managed using API version 2019-05-06.

You can update flat indexes to the new format with the following steps using API version [2017-11-11-Preview](#):

1. Perform a GET request to retrieve your index. If it's already in the new format, you're done.
2. Translate the index from the flat format to the new format. You have to write code for this task since there's no sample code available at the time of this writing.

3. Perform a PUT request to update the index to the new format. Avoid changing any other details of the index, such as the searchability/filterability of fields, because changes that affect the physical expression of existing index isn't allowed by the Update Index API.

 Note

It isn't possible to manage indexes created with the old "flat" format from the Azure portal. Upgrade your indexes from the "flat" representation to the "tree" representation at your earliest convenience.

Control plane upgrades

Applies to: 2014-07-31-Preview, 2015-02-28, and 2015-08-19

The `listQueryKeys` GET request on older Search Management API versions is now deprecated. We recommend migrating to the most recent stable control plane API version to use the [listQueryKeys POST request](#).

1. In existing code, change the `api-version` parameter to the most recent version (`2025-05-01`).
2. Reframe the request from `GET` to `POST`:

HTTP

POST

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Search/searchServices/{searchServiceName}/listQueryKeys?api-version=2025-05-01
Authorization: Bearer {{token}}
```

3. If you're using an Azure SDK, it's recommended that you upgrade to the latest version.

Next steps

Review the Search REST API reference documentation. If you encounter problems, ask us for help on [Stack Overflow](#) or [contact support](#).

[Search service REST API Reference](#)

Upgrade to Azure AI Search .NET SDK version 11

05/29/2025

If your search solution is built on the [Azure SDK for .NET](#), this article helps you migrate your code from earlier versions of [Microsoft.Azure.Search](#) to version 11, the new [Azure.Search.Documents](#) client library. Version 11 is a fully redesigned client library, released by the Azure SDK development team (previous versions were produced by the Azure AI Search development team).

All features from version 10 are implemented in version 11. Key differences include:

- One package ([Azure.Search.Documents](#)) instead of four
- Three clients instead of two: `SearchClient`, `SearchIndexClient`, `SearchIndexerClient`
- Naming differences across a range of APIs and small structural differences that simplify some tasks

The client library's [Change Log](#) has an itemized list of updates. You can review a [summarized version](#) in this article.

All C# code samples and snippets in the Azure AI Search product documentation have been revised to use the new [Azure.Search.Documents](#) client library.

Why upgrade?

The benefits of upgrading are summarized as follows:

- New features are added to [Azure.Search.Documents](#) only. The previous version, [Microsoft.Azure.Search](#), is now retired. Updates to deprecated libraries are limited to high priority bug fixes only.
- Consistency with other Azure client libraries. [Azure.Search.Documents](#) takes a dependency on [Azure.Core](#) and [System.Text.Json](#), and follows conventional approaches for common tasks such as client connections and authorization.

[Microsoft.Azure.Search](#) is officially retired. If you're using an old version, we recommend upgrading to the next higher version, repeating the process in succession until you reach version 11 and [Azure.Search.Documents](#). An incremental upgrade strategy makes it easier to find and fix blocking issues. See [Previous version docs](#) for guidance.

Package comparison

Version 11 consolidates and simplifies package management so that there are fewer to manage.

[] [Expand table](#)

Version 10 and earlier	Version 11
Microsoft.Azure.Search	Azure.Search.Documents package
Microsoft.Azure.Search.Service	
Microsoft.Azure.Search.Data	
Microsoft.Azure.Search.Common	

Client comparison

Where applicable, the following table maps the client libraries between the two versions.

[] [Expand table](#)

Client operations	Microsoft.Azure.Search (v10)	Azure.Search.Documents (v11)
Targets the documents collection of an index (queries and data import)	SearchIndexClient	SearchClient
Targets index-related objects (indexes, analyzers, synonym maps)	SearchServiceClient	SearchIndexClient
Targets indexer-related objects (indexers, data sources, skillsets)	SearchServiceClient	SearchIndexerClient (new)

✖ Caution

Notice that SearchIndexClient exists in both versions, but targets different operations. In version 10, SearchIndexClient creates indexes and other objects. In version 11, SearchIndexClient works with existing indexes, targeting the documents collection with query and data ingestion APIs. To avoid confusion when updating code, be mindful of the order in which client references are updated. Following the sequence in [Steps to upgrade](#) should help mitigate any string replacement issues.

Naming and other API differences

Besides the client differences (noted previously and thus omitted here), multiple other APIs have been renamed and in some cases redesigned. Class name differences are summarized in

the following sections. This list isn't exhaustive but it does group API changes by task, which can be helpful for revisions on specific code blocks. For an itemized list of API updates, see the [change log](#) for `Azure.Search.Documents` on GitHub.

Authentication and encryption

 [Expand table](#)

Version 10	Version 11 equivalent
SearchCredentials	AzureKeyCredential
EncryptionKey (Undocumented in API reference. Support for this API transitioned to generally available in v10, but was only available in the preview SDK)	SearchResourceEncryptionKey

Indexes, analyzers, synonym maps

 [Expand table](#)

Version 10	Version 11 equivalent
Index	SearchIndex
Field	SearchField
DataType	SearchFieldDataType
ItemError	SearchIndexerError
Analyzer	LexicalAnalyzer (also, <code>AnalyzerName</code> to <code>LexicalAnalyzerName</code>)
AnalyzeRequest	AnalyzeTextOptions
StandardAnalyzer	LuceneStandardAnalyzer
StandardTokenizer	LuceneStandardTokenizer (also, <code>StandardTokenizerV2</code> to <code>LuceneStandardTokenizerV2</code>)
TokenInfo	AnalyzedTokenInfo
Tokenizer	LexicalTokenizer (also, <code>TokenizerName</code> to <code>LexicalTokenizerName</code>)
SynonymMap.Format	None. Remove references to <code>Format</code> .

Field definitions are streamlined: `SearchableField`, `SimpleField`, `ComplexField` are new APIs for creating field definitions.

Indexers, datasources, skillsets

[+] [Expand table](#)

Version 10	Version 11 equivalent
Indexer	SearchIndexer
DataSource	SearchIndexerDataSourceConnection
Skill	SearchIndexerSkill
Skillset	SearchIndexerSkillset
DataSourceType	SearchIndexerDataSourceType

Data import

[+] [Expand table](#)

Version 10	Version 11 equivalent
IndexAction	IndexDocumentsAction
IndexBatch	IndexDocumentsBatch
IndexBatchException.FindFailedActionsToRetry()	SearchIndexingBufferedSender

Query requests and responses

[+] [Expand table](#)

Version 10	Version 11 equivalent
DocumentsOperationsExtensions.SearchAsync	SearchClient.SearchAsync
DocumentSearchResult	SearchResult or SearchResults, depending on whether the result is a single document or multiple.
DocumentSuggestResult	SuggestResults
SearchParameters	SearchOptions
SuggestParameters	SuggestOptions
SearchParameters.Filter	SearchFilter (a new class for constructing OData filter expressions)

JSON serialization

By default, the Azure SDK uses [System.Text.Json](#) for JSON serialization, relying on the capabilities of those APIs to handle text transformations previously implemented through a native [SerializePropertyNamesAsCamelCaseAttribute](#) class, which has no counterpart in the new library.

To serialize property names into camelCase, you can use the [JsonPropertyNameAttribute](#) (similar to [this example](#)).

Alternatively, you can set a [JsonNamingPolicy](#) provided in [JsonSerializerOptions](#). The following System.Text.Json code example, taken from the [Microsoft.Azure.Core.Spatial](#) readme demonstrates the use of camelCase without having to attribute every property:

C#

```
// Get the Azure AI Search service endpoint and read-only API key.
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
AzureKeyCredential credential = new
AzureKeyCredential(Environment.GetEnvironmentVariable("SEARCH_API_KEY"));

// Create serializer options with our converter to deserialize geographic points.
JsonSerializerOptions serializerOptions = new JsonSerializerOptions
{
    Converters =
    {
        new MicrosoftSpatialGeoJsonConverter()
    },
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
};

SearchClientOptions clientOptions = new SearchClientOptions
{
    Serializer = new JsonObjectSerializer(serializerOptions)
};

SearchClient client = new SearchClient(endpoint, "mountains", credential,
clientOptions);
Response<SearchResults<Mountain>> results = client.Search<Mountain>("Rainier");
```

If you're using [Newtonsoft.Json](#) for JSON serialization, you can pass in global naming policies using similar attributes, or by using properties on [JsonSerializerSettings](#). For an example equivalent to the previous one, see the [Deserializing documents example](#) in the [Newtonsoft.Json](#) readme.

Inside v11

Each version of an Azure AI Search client library targets a corresponding version of the REST API. The REST API is considered foundational to the service, with individual SDKs wrapping a version of the REST API. As a .NET developer, it can be helpful to review the more verbose [REST API documentation](#) for more in depth coverage of specific objects or operations. Version 11 targets the [2020-06-30 search service specification](#).

Version 11.0 fully supports the following objects and operations:

- Index creation and management
- Synonym map creation and management
- Indexer creation and management
- Indexer data source creation and management
- Skillset creation and management
- All query types and syntax

Version 11.1 additions ([change log](#) details):

- [FieldBuilder](#) (added in 11.1)
- [Serializer property](#) (added in 11.1) to support custom serialization

Version 11.2 additions ([change log](#) details):

- [EncryptionKey](#) property added indexers, data sources, and skillsets
- [IndexingParameters.IndexingParametersConfiguration](#) property support
- [Geospatial types](#) are natively supported in [FieldBuilder](#). [SearchFilter](#) can encode geometric types from Microsoft.Spatial without an explicit assembly dependency.

You can also continue to explicitly declare a dependency on [Microsoft.Spatial](#). Examples of this technique are available for [System.Text.Json](#) and [Newtonsoft.Json](#).

Version 11.3 additions ([change log](#) details):

- [KnowledgeStore](#)
- Added support for Azure.Core.GeoJson types in [SearchDocument](#), [SearchFilter](#) and [FieldBuilder](#).
- Added EventSource based logging. Event source name is Azure-Search-Documents. Current set of events are focused on tuning batch sizes for [SearchIndexingBufferedSender](#).
- Added [CustomEntityLookupSkill](#) and [DocumentExtractionSkill](#). Added DefaultCountryHint in [LanguageDetectionSkill](#).

Before upgrading

- Quickstarts, tutorials, and [C# samples](#) have been updated to use the `Azure.Search.Documents` package. We recommend reviewing the samples and walkthroughs to learn about the new APIs before embarking on a migration exercise.
- [How to use Azure.Search.Documents](#) introduces the most commonly used APIs. Even knowledgeable users of Azure AI Search might want to review this introduction to the new library as a precursor to migration.

Steps to upgrade

The following steps get you started on a code migration by walking through the first set of required tasks, especially regarding client references.

1. Install the [Azure.Search.Documents package](#) by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.
2. Replace using directives for `Microsoft.Azure.Search` with the following using statements:

```
C#
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.Models;
```

3. For classes that require JSON serialization, replace `using Newtonsoft.Json` with `using System.Text.Json.Serialization`.
4. Revise client authentication code. In previous versions, you would use properties on the client object to set the API key (for example, the `SearchServiceClient.Credentials` property). In the current version, use the `AzureKeyCredential` class to pass the key as a credential, so that if needed, you can update the API key without creating new client objects.

Client properties have been streamlined to just `Endpoint`, `ServiceName`, and `IndexName` (where appropriate). The following example uses the system `Uri` class to provide the endpoint and the `Environment` class to read in the key value:

```
C#
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
AzureKeyCredential credential = new AzureKeyCredential(
```

```
Environment.GetEnvironmentVariable("SEARCH_API_KEY"));
SearchIndexClient indexClient = new SearchIndexClient(endpoint, credential);
```

5. Add new client references for indexer-related objects. If you're using indexers, datasources, or skillsets, change the client references to [SearchIndexClient](#). This client is new in version 11 and has no antecedent.
6. Revise collections and lists. In the new SDK, all lists are read-only to avoid downstream issues if the list happens to contain null values. The code change is to add items to a list. For example, instead of assigning strings to a Select property, you would add them as follows:

```
C#  
  
var options = new SearchOptions
{
    SearchMode = SearchMode.All,
    IncludeTotalCount = true
};

// Select fields to return in results.
options.Select.Add("HotelName");
options.Select.Add("Description");
options.Select.Add("Tags");
options.Select.Add("Rooms");
options.Select.Add("Rating");
options.Select.Add("LastRenovationDate");
```

Select, Facets, SearchFields, SourceFields, ScoringParameters, and OrderBy are all lists that now need to be reconstructed.

7. Update client references for queries and data import. Instances of [SearchIndexClient](#) should be changed to [SearchClient](#). To avoid name confusion, make sure you catch all instances before proceeding to the next step.
8. Update client references for index, synonym map, and analyzer objects. Instances of [SearchServiceClient](#) should be changed to [SearchIndexClient](#).
9. For the remainder of your code, update classes, methods, and properties to use the APIs of the new library. The [naming differences](#) section is a place to start but you can also review the [change log ↴](#).

If you have trouble finding equivalent APIs, we suggest logging an issue on <https://github.com/MicrosoftDocs/azure-docs/issues> so that we can improve the documentation or investigate the problem.

10. Rebuild the solution. After fixing any build errors or warnings, you can make additional changes to your application to take advantage of [new functionality](#).

Breaking changes

Given the sweeping changes to libraries and APIs, an upgrade to version 11 is non-trivial and constitutes a breaking change in the sense that your code will no longer be backward compatible with version 10 and earlier. For a thorough review of the differences, see the [change log ↗](#) for `Azure.Search.Documents`.

In terms of service version updates, where code changes in version 11 relate to existing functionality (and not just a refactoring of the APIs), you'll find the following behavior changes:

- [BM25 ranking algorithm](#) replaces the previous ranking algorithm with newer technology. New services use this algorithm automatically. For existing services, you must set parameters to use the new algorithm.
- [Ordered results](#) for null values have changed in this version, with null values appearing first if the sort is `asc` and last if the sort is `desc`. If you wrote code to handle how null values are sorted, you should review and potentially remove that code if it's no longer necessary.

Due to these behavior changes, it's likely that there are slight variations in ranked results.

Next steps

- [How to use Azure.Search.Documents in a C# .NET Application](#)
- [Tutorial: Add search to web apps](#)
- [Azure.Search.Documents package ↗](#)
- [Samples on GitHub ↗](#)
- [Azure.Search.Document API reference](#)

How to use Azure.Search.Documents in a .NET application

Article • 04/15/2025

This article explains how to create and manage search objects using C# and the [Azure.Search.Documents](#) client library in the Azure SDK for .NET.

You can use this library for data plane operations, including:

- Create and manage search indexes, data sources, indexers, skillsets, and synonym maps
- Load and manage search documents in an index
- Execute queries, all without having to deal with the details of HTTP and JSON
- Invoke and manage AI enrichment (skillsets) and outputs

The library is distributed as a single [NuGet package](#) that includes all APIs used for programmatic access to a search service.

The client library defines classes like `SearchIndex`, `SearchField`, and `SearchDocument`, as well as operations like `SearchIndexClient.CreateIndex` and `SearchClient.Search` on the `SearchIndexClient` and `SearchClient` classes. These classes are organized into the following namespaces:

- [Azure.Search.Documents](#)
- [Azure.Search.Documents.Indexes](#)
- [Azure.Search.Documents.Indexes.Models](#)
- [Azure.Search.Documents.Models](#)

The [Azure.Search.Documents](#) client library doesn't provide [service management operations](#), such as creating and scaling search services and managing API keys. If you need to manage your search resources from a .NET application, use the [Azure.ResourceManager.Search](#) library in the Azure SDK for .NET.

SDK requirements

- Visual Studio 2019 or later.
- [Azure AI Search](#)
- Download the [NuGet package](#) using **Tools > NuGet Package Manager > Manage NuGet Packages for Solution** in Visual Studio. Search for the package name `Azure.Search.Documents`.

The Azure SDK for .NET conforms to [.NET Standard 2.0](#).

Example application

This article *teaches by example*, relying on the [DotNetHowTo](#) code example on GitHub to illustrate fundamental concepts in Azure AI Search, and how to create, load, and query a search index.

For the rest of this article, assume a new index named *hotels*, populated with a few documents, with several queries that match on results.

The following example shows the main program, with the overall flow:

C#

```
// This sample shows how to delete, create, upload documents and query an index
static void Main(string[] args)
{
    IConfigurationBuilder builder = new
    ConfigurationBuilder().AddJsonFile("appsettings.json");
    IConfigurationRoot configuration = builder.Build();

    SearchIndexClient indexClient = CreateSearchIndexClient(configuration);

    string indexName = configuration["SearchIndexName"];

    Console.WriteLine("{0}", "Deleting index...\\n");
    DeleteIndexIfExists(indexName, indexClient);

    Console.WriteLine("{0}", "Creating index...\\n");
    CreateIndex(indexName, indexClient);

    SearchClient searchClient = indexClient.GetSearchClient(indexName);

    Console.WriteLine("{0}", "Uploading documents...\\n");
    UploadDocuments(searchClient);

    SearchClient indexClientForQueries = CreateSearchClientForQueries(indexName,
configuration);

    Console.WriteLine("{0}", "Run queries...\\n");
    RunQueries(indexClientForQueries);

    Console.WriteLine("{0}", "Complete. Press any key to end application...\\n");
    Console.ReadKey();
}
```

Next is a partial screenshot of the output, assuming you run this application with a valid service name and API keys:

```
C:\Program Files\dotnet\dotnet.exe
Deleting index...
Creating index...
Uploading documents...
Waiting for documents to be indexed...
Query 1: Search for 'motel'. Return only the HotelName in results:
Name: Twin Dome Motel
Name: Secret Point Motel

Query 2: Apply a filter to find hotels with rooms cheaper than $100 per night, areturning the HotelId and Description:
HotelId: 2
Description: The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to the highest architectural standards to create a modern, functional and first-class hotel in which art and unique historical elements coexist with the most modern comforts.

HotelId: 1
Description: The hotel is ideally located on the main commercial artery of the city in the heart of New York. A few minutes away is Time's Square and the historic centre of the city, as well as other places of interest that make New York one of America's most attractive and cosmopolitan cities.
```

Client types

The client library uses three client types for various operations: [SearchIndexClient](#) to create, update, or delete indexes, [SearchClient](#) to load or query an index, and [SearchIndexerClient](#) to work with indexers and skillsets. This article focuses on the first two.

At a minimum, all of the clients require the service name or endpoint, and an API key. It's common to provide this information in a configuration file, similar to what you find in the `appsettings.json` file of the [DotNetHowTo sample application](#). To read from the configuration file, add `using Microsoft.Extensions.Configuration;` to your program.

The following statement creates the index client used to create, update, or delete indexes. It takes a service endpoint and admin API key.

```
C#
private static SearchIndexClient CreateSearchIndexClient(IConfigurationRoot configuration)
{
    string searchServiceEndPoint = configuration["YourSearchServiceEndPoint"];
    string adminApiKey = configuration["YourSearchServiceAdminApiKey"];

    SearchIndexClient indexClient = new SearchIndexClient(new
        Uri(searchServiceEndPoint), new AzureKeyCredential(adminApiKey));
    return indexClient;
}
```

The next statement creates the search client used to load documents or run queries.

`SearchClient` requires an index. You need an admin API key to load documents, but you can

use a query API key to run queries.

C#

```
string indexName = configuration["SearchIndexName"];  
  
private static SearchClient CreateSearchClientForQueries(string indexName,  
IConfigurationRoot configuration)  
{  
    string searchServiceEndPoint = configuration["YourSearchServiceEndPoint"];  
    string queryApiKey = configuration["YourSearchServiceQueryApiKey"];  
  
    SearchClient searchClient = new SearchClient(new Uri(searchServiceEndPoint),  
indexName, new AzureKeyCredential(queryApiKey));  
    return searchClient;  
}
```

(!) Note

If you provide an invalid key for the import operation (for example, a query key where an admin key was required), the `SearchClient` throws a `CloudException` with the error message *Forbidden* the first time you call an operation method on it. If this happens to you, double-check the API key.

Delete the index

In the early stages of development, you might want to include a `DeleteIndex` statement to delete a work-in-progress index so that you can recreate it with an updated definition. Sample code for Azure AI Search often includes a deletion step so that you can rerun the sample.

The following line calls `DeleteIndexIfExists`:

C#

```
Console.WriteLine("{0}", "Deleting index...\\n");  
DeleteIndexIfExists(indexName, indexClient);
```

This method uses the given `SearchIndexClient` to check if the index exists, and if so, deletes it:

C#

```
private static void DeleteIndexIfExists(string indexName, SearchIndexClient  
indexClient)  
{  
    try
```

```
{  
    if (indexClient.GetIndex(indexName) != null)  
    {  
        indexClient.DeleteIndex(indexName);  
    }  
}  
catch (RequestFailedException e) when (e.Status == 404)  
{  
    // Throw an exception if the index name isn't found  
    Console.WriteLine("The index doesn't exist. No deletion occurred.");
```

ⓘ Note

The example code in this article uses the synchronous methods for simplicity, but you should use the asynchronous methods in your own applications to keep them scalable and responsive. For example, in the previous method, you could use [DeleteIndexAsync](#) instead of [DeleteIndex](#).

Create an index

You can use [SearchIndexClient](#) to create an index.

The following method creates a new [SearchIndex](#) object with a list of [SearchField](#) objects that define the schema of the new index. Each field has a name, data type, and several attributes that define its search behavior.

Fields can be defined from a model class using [FieldBuilder](#). The [FieldBuilder](#) class uses reflection to create a list of [SearchField](#) objects for the index by examining the public properties and attributes of the given [Hotel](#) model class. We'll take a closer look at the [Hotel](#) class later on.

C#

```
private static void CreateIndex(string indexName, SearchIndexClient indexClient)  
{  
    FieldBuilder fieldBuilder = new FieldBuilder();  
    var searchFields = fieldBuilder.Build(typeof(Hotel));  
  
    var definition = new SearchIndex(indexName, searchFields);  
  
    indexClient.CreateOrUpdateIndex(definition);  
}
```

Besides fields, you could also add scoring profiles, suggesters, or CORS options to the index (these parameters are omitted from the sample for brevity). You can find more information about the `SearchIndex` object and its constituent parts in the [SearchIndex properties list](#), as well as in the [REST API reference](#).

! Note

You can always create the list of `Field` objects directly instead of using `FieldBuilder` if needed. For example, you might not want to use a model class or you might need to use an existing model class that you don't want to modify by adding attributes.

Call CreateIndex in Main()

`Main` creates a new *hotels* index by calling the preceding method:

C#

```
Console.WriteLine("{0}", "Creating index...\n");
CreateIndex(indexName, indexClient);
```

Use a model class for data representation

The DotNetHowTo sample uses model classes for the [Hotel](#), [Address](#), and [Room](#) data structures. `Hotel` references `Address`, a single level complex type (a multi-part field), and `Room` (a collection of multi-part fields).

You can use these types to create and load the index, and to structure the response from a query:

C#

```
// Use-case: <Hotel> in a field definition
FieldBuilder fieldBuilder = new FieldBuilder();
var searchFields = fieldBuilder.Build(typeof(Hotel));

// Use-case: <Hotel> in a response
private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach ( SearchResult<Hotel> result in searchResults.GetResults())
    {
        Console.WriteLine(result.Document);
    }
}
```

```
        Console.WriteLine();
    }
```

An alternative approach is to add fields to an index directly. The following example shows just a few fields.

C#

```
SearchIndex index = new SearchIndex(indexName)
{
    Fields =
    {
        new SimpleField("hotelId", SearchFieldDataType.String) { IsKey = true, IsFilterable = true, IsSortable = true },
        new SearchableField("hotelName") { IsFilterable = true, IsSortable = true },
        new SearchableField("hotelCategory") { IsFilterable = true, IsSortable = true },
        new SimpleField("baseRate", SearchFieldDataType.Int32) { IsFilterable = true, IsSortable = true },
        new SimpleField("lastRenovationDate",
SearchFieldDataType.DateTimeOffset) { IsFilterable = true, IsSortable = true }
    }
};
```

Field definitions

Your data model in .NET and its corresponding index schema should support the search experience you'd like to give to your end user. Each top level object in .NET, such as a search document in a search index, corresponds to a search result you would present in your user interface. For example, in a hotel search application, your end users might want to search by hotel name, features of the hotel, or the characteristics of a particular room.

Within each class, a field is defined with a data type and attributes that determine how it's used. The name of each public property in each class maps to a field with the same name in the index definition.

Take a look at the following snippet that pulls several field definitions from the Hotel class. Notice that `Address` and `Rooms` are C# types with their own class definitions (refer to the sample code if you want to view them). Both are complex types. For more information, see [How to model complex types](#).

C#

```
public partial class Hotel
{
```

```

[SimpleField(IsKey = true, IsFilterable = true)]
public string HotelId { get; set; }

[SearchableField(IsSortable = true)]
public string HotelName { get; set; }

[SearchableField(AnalyzerName = LexicalAnalyzerName.Values.EnLucene)]
public string Description { get; set; }

[SearchableField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
public string Category { get; set; }

[JsonIgnore]
public bool? SmokingAllowed => (Rooms != null) ? Array.Exists(Rooms, element => element.SmokingAllowed == true) : (bool?)null;

[SearchableField]
public Address Address { get; set; }

public Room[] Rooms { get; set; }

```

Choose a field class

When defining fields, you can use the base `SearchField` class, or you can use derivative helper models that serve as *templates*, with pre-configured properties.

Exactly one field in your index must serve as the document key (`IsKey = true`). It must be a string, and it must uniquely identify each document. It's also required to have `IsHidden = true`, which means it can't be visible in search results.

[] [Expand table](#)

Field type	Description and usage
<code>SearchField</code>	Base class, with most properties set to null, except <code>Name</code> which is required, and <code>AnalyzerName</code> which defaults to standard Lucene.
<code>SimpleField</code>	Helper model. Can be any data type, is always non-searchable (it's ignored for full text search queries), and is retrievable (it's not hidden). Other attributes are off by default, but can be enabled. You might use a <code>SimpleField</code> for document IDs or fields used only in filters, facets, or scoring profiles. If so, be sure to apply any attributes that are necessary for the scenario, such as <code>IsKey = true</code> for a document ID. For more information, see SimpleFieldAttribute.cs in source code.
<code>SearchableField</code>	Helper model. Must be a string, and is always searchable and retrievable. Other attributes are off by default, but can be enabled. Because this field type is searchable, it supports synonyms and the full complement of analyzer properties. For more information, see the SearchableFieldAttribute.cs in source code.

Whether you use the basic `SearchField` API or either one of the helper models, you must explicitly enable filter, facet, and sort attributes. For example, `IsFilterable`, `IsSortable`, and `IsFacetable` must be explicitly attributed, as shown in the previous sample.

Add field attributes

Notice how each field is decorated with attributes such as `IsFilterable`, `IsSortable`, `IsKey`, and `AnalyzerName`. These attributes map directly to the [corresponding field attributes in an Azure AI Search index](#). The `FieldBuilder` class uses these properties to construct field definitions for the index.

Field type mapping

The .NET types of the properties map to their equivalent field types in the index definition. For example, the `Category` string property maps to the `category` field, which is of type `Edm.String`. There are similar type mappings between `bool?`, `Edm.Boolean`, `DateTimeOffset?`, and `Edm.DateTimeOffset` and so on.

Did you happen to notice the `SmokingAllowed` property?

C#

```
[JsonIgnore]
public bool? SmokingAllowed => (Rooms != null) ? Array.Exists(Rooms, element =>
element.SmokingAllowed == true) : (bool?)null;
```

The `JsonIgnore` attribute on this property tells the `FieldBuilder` to not serialize it to the index as a field. This is a great way to create client-side calculated properties that you can use as helpers in your application. In this case, the `SmokingAllowed` property reflects whether any `Room` in the `Rooms` collection allows smoking. If all are false, it indicates that the entire hotel doesn't allow smoking.

Load an index

The next step in `Main` populates the newly created `hotels` index. This index population is done in the following method: (Some code replaced with `...` for illustration purposes. See the full sample solution for the full data population code.)

C#

```
private static void UploadDocuments(SearchClient searchClient)
{
    IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                HotelName = "Stay-Kay City Hotel",
                ...
                Address = new Address()
                {
                    StreetAddress = "677 5th Ave",
                    ...
                },
                Rooms = new Room[]
                {
                    new Room()
                    {
                        Description = "Budget Room, 1 Queen Bed (Cityside)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Budget Room, 1 King Bed (Mountain View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Deluxe Room, 2 Double Beds (City View)",
                        ...
                    }
                }
            },
            IndexDocumentsAction.Upload(
                new Hotel()
                {
                    HotelId = "2",
                    HotelName = "Old Century Hotel",
                    ...
                },
                Rooms = new Room[]
                {
                    new Room()
                    {
                        Description = "Suite, 2 Double Beds (Mountain View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Standard Room, 1 Queen Bed (City View)",
                        ...
                    }
                }
            )
        )
    );
}
```

```
        },
        new Room()
    {
        Description = "Budget Room, 1 King Bed (Waterfront View)",
        ...
    }
},
IndexDocumentsAction.Upload(
    new Hotel()
{
    HotelId = "3",
    HotelName = "Gastronomic Landscape Hotel",
    ...
    Address = new Address()
    {
        StreetAddress = "3393 Peachtree Rd",
        ...
    },
    Rooms = new Room[]
    {
        new Room()
        {
            Description = "Standard Room, 2 Queen Beds (Amenities)",
            ...
        },
        new Room()
        {
            Description = "Standard Room, 2 Double Beds (Waterfront
View)",
            ...
        },
        new Room()
        {
            Description = "Deluxe Room, 2 Double Beds (Cityside)",
            ...
        }
    }
}
);

try
{
    IndexDocumentsResult result = searchClient.IndexDocuments(batch);
}
catch (Exception)
{
    // Sometimes when your Search service is under load, indexing will fail
    // for some of the documents in
    // the batch. Depending on your application, you can take compensating
    // actions like delaying and
    // retrying. For this simple demo, we just log the failed document keys
    // and continue.
    Console.WriteLine("Failed to index some of the documents: {0}");
}
```

```
Console.WriteLine("Waiting for documents to be indexed... \n");
Thread.Sleep(2000);
```

This method has four parts. The first creates an array of three `Hotel` objects each with three `Room` objects that serve as our input data to upload to the index. This data is hard-coded for simplicity. In an actual application, data likely comes from an external data source such as a SQL database.

The second part creates an `IndexDocumentsBatch` containing the documents. You specify the operation you want to apply to the batch at the time you create it, in this case by calling `IndexDocumentsAction.Upload`. The batch is then uploaded to the Azure AI Search index by the `IndexDocuments` method.

Note

In this example, you're just uploading documents. If you wanted to merge changes into existing documents or delete documents, you could create batches by calling `IndexDocumentsAction.Merge`, `IndexDocumentsAction.MergeOrUpload`, or `IndexDocumentsAction.Delete` instead. You can also mix different operations in a single batch by calling `IndexBatch.New`, which takes a collection of `IndexDocumentsAction` objects, each of which tells Azure AI Search to perform a particular operation on a document. You can create each `IndexDocumentsAction` with its own operation by calling the corresponding method such as `IndexDocumentsAction.Merge`, `IndexAction.Upload`, and so on.

The third part of this method is a catch block that handles an important error case for indexing. If your search service fails to index some of the documents in the batch, a `RequestFailedException` is thrown. An exception can happen if you're indexing documents while your service is under heavy load. **We strongly recommend explicitly handling this case in your code.** You can delay and then retry indexing the documents that failed, or you can log and continue like the sample does, or you can do something else depending on your application's data consistency requirements. An alternative is to use `SearchIndexingBufferedSender` for intelligent batching, automatic flushing, and retries for failed indexing actions. For more context, see the [SearchIndexingBufferedSender example ↗](#).

Finally, the `UploadDocuments` method delays for two seconds. Indexing happens asynchronously in your search service, so the sample application needs to wait a short time to ensure that the documents are available for searching. Delays like this are typically only necessary in demos, tests, and sample applications.

Call UploadDocuments in Main()

The following code snippet sets up an instance of `SearchClient` using the `GetSearchClient` method of `indexClient`. The `indexClient` uses an admin API key on its requests, which is required for loading or refreshing documents.

An alternate approach is to call `SearchClient` directly, passing in an admin API key on `AzureKeyCredential`.

C#

```
SearchClient searchClient = indexClient.GetSearchClient(indexName);

Console.WriteLine("{0}", "Uploading documents...\n");
UploadDocuments(searchClient);
```

Run queries

First, set up a `SearchClient` that reads the service endpoint and query API key from `appsettings.json`:

C#

```
private static SearchClient CreateSearchClientForQueries(string indexName,
IConfigurationRoot configuration)
{
    string searchServiceEndPoint = configuration["YourSearchServiceEndPoint"];
    string queryApiKey = configuration["YourSearchServiceQueryApiKey"];

    SearchClient searchClient = new SearchClient(new Uri(searchServiceEndPoint),
indexName, new AzureKeyCredential(queryApiKey));
    return searchClient;
}
```

Second, define a method that sends a query request.

Each time the method executes a query, it creates a new `SearchOptions` object. This object is used to specify additional options for the query such as sorting, filtering, paging, and faceting. In this method, we're setting the `Filter`, `Select`, and `OrderBy` property for different queries. For more information about the search query expression syntax, [Simple query syntax](#).

The next step is query execution. Running the search is done using the `SearchClient.Search` method. For each query, pass the search text to use as a string (or `"*"` if there's no search text), plus the search options created earlier. We also specify `Hotel` as the type parameter for

`SearchClient.Search`, which tells the SDK to deserialize documents in the search results into objects of type `Hotel`.

C#

```
private static void RunQueries(SearchClient searchClient)
{
    SearchOptions options;
    SearchResults<Hotel> results;

    Console.WriteLine("Query 1: Search for 'motel'. Return only the HotelName in results:\n");

    options = new SearchOptions();
    options.Select.Add("HotelName");

    results = searchClient.Search<Hotel>("motel", options);

    WriteDocuments(results);

    Console.Write("Query 2: Apply a filter to find hotels with rooms cheaper than $100 per night, ");
    Console.WriteLine("returning the HotelId and Description:\n");

    options = new SearchOptions()
    {
        Filter = "Rooms/any(r: r/BaseRate lt 100)"
    };
    options.Select.Add("HotelId");
    options.Select.Add("Description");

    results = searchClient.Search<Hotel>("*", options);

    WriteDocuments(results);

    Console.Write("Query 3: Search the entire index, order by a specific field (lastRenovationDate) ");
    Console.Write("in descending order, take the top two results, and show only hotelName and ");
    Console.WriteLine("lastRenovationDate:\n");

    options =
        new SearchOptions()
    {
        Size = 2
    };
    options.OrderBy.Add("LastRenovationDate desc");
    options.Select.Add("HotelName");
    options.Select.Add("LastRenovationDate");

    results = searchClient.Search<Hotel>("*", options);

    WriteDocuments(results);
```

```

Console.WriteLine("Query 4: Search the HotelName field for the term
'hotel':\n");

options = new SearchOptions();
options.SearchFields.Add("HotelName");

//Adding details to select, because "Location" isn't supported yet when
deserializing search result to "Hotel"
options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Description");
options.Select.Add("Category");
options.Select.Add("Tags");
options.Select.Add("ParkingIncluded");
options.Select.Add("LastRenovationDate");
options.Select.Add("Rating");
options.Select.Add("Address");
options.Select.Add("Rooms");

results = searchClient.Search<Hotel>("hotel", options);

WriteDocuments(results);
}

```

Third, define a method that writes the response, printing each document to the console:

```

C#

private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.GetResults())
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}

```

Call RunQueries in Main()

```

C#

SearchClient indexClientForQueries = CreateSearchClientForQueries(indexName,
configuration);

Console.WriteLine("{0}", "Running queries...\n");
RunQueries(indexClientForQueries);

```

Explore query constructs

Let's take a closer look at each of the queries in turn. Here's the code to execute the first query:

```
C#  
  
options = new SearchOptions();  
options.Select.Add("HotelName");  
  
results = searchClient.Search<Hotel>("motel", options);  
  
WriteDocuments(results);
```

In this case, we're searching the entire index for the word *motel* in any searchable field and we only want to retrieve the hotel names, as specified by the `Select` option. Here are the results:

```
Output  
  
Name: Stay-Kay City Hotel  
  
Name: Old Century Hotel
```

In the second query, use a filter to select rooms with a nightly rate of less than \$100. Return only the hotel ID and description in the results:

```
C#  
  
options = new SearchOptions()  
{  
    Filter = "Rooms/any(r: r/BaseRate lt 100)"  
};  
options.Select.Add("HotelId");  
options.Select.Add("Description");  
  
results = searchClient.Search<Hotel>("*", options);
```

This query uses an OData `$filter` expression, `Rooms/any(r: r/BaseRate lt 100)`, to filter the documents in the index. It uses the [any operator](#) to apply the 'BaseRate lt 100' to every item in the Rooms collection. For more information, see [OData filter syntax](#).

In the third query, find the top two hotels that have been most recently renovated, and show the hotel name and last renovation date. Here's the code:

```
C#  
  
options =  
    new SearchOptions()
```

```

    {
        Size = 2
    };
options.OrderBy.Add("LastRenovationDate desc");
options.Select.Add("HotelName");
options.Select.Add("LastRenovationDate");

results = searchClient.Search<Hotel>("*", options);

WriteDocuments(results);

```

In the last query, find all hotels names that match the word *hotel*:

C#

```

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Description");
options.Select.Add("Category");
options.Select.Add("Tags");
options.Select.Add("ParkingIncluded");
options.Select.Add("LastRenovationDate");
options.Select.Add("Rating");
options.Select.Add("Address");
options.Select.Add("Rooms");

results = searchClient.Search<Hotel>("hotel", options);

WriteDocuments(results);

```

This section concludes this introduction to the .NET SDK, but don't stop here. The next section suggests other resources for learning more about programming with Azure AI Search.

Related content

- Browse the API reference documentation for [Azure.Search.Documents](#) and [REST API](#)
- Browse other code samples based on Azure.Search.Documents in [azure-search-dotnet-samples](#) and [search-dotnet-getting-started](#)
- Review [naming conventions](#) to learn the rules for naming various objects
- Review [supported data types](#)

Upgrade versions of the Azure Search .NET Management SDK

Article • 02/24/2025

This article points you to libraries in the Azure SDK for .NET for managing a search service. These libraries provide the APIs used to create, configure, and delete search services. They also provide APIs used to adjust capacity, manage API keys, and configure network security.

Management SDKs target a specific version of the Management REST API. Release notes for each library indicate which REST API version is the target for each package. For more information about concepts and operations, see [Search Management \(REST\)](#).

Versions

The following table lists the client libraries used to provision a search service.

[+] Expand table

Namespace	Version	Status	Change log
Azure.ResourceManager.Search	Package versions ↗	Current	Change Log ↗
Microsoft.Azure.Management.Search	Package versions ↗	Deprecated	Release notes ↗

Checklist for upgrade

1. Review the [change log ↗](#) for updates to the library.
2. In your application code, delete the reference to `Microsoft.Azure.Management.Search` and its dependencies.
3. Add a reference for `Azure.ResourceManager.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.
4. Once NuGet has downloaded the new packages and their dependencies, replace the API calls.

Next steps

If you encounter problems, the best forum for posting questions is [Stack Overflow](#). If you find a bug, you can file an issue in the [Azure .NET SDK GitHub repository](#). Make sure to label your issue title with *search*.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Manage concurrency in Azure AI Search

08/22/2025

When managing Azure AI Search resources such as indexes and data sources, it's important to update resources safely, especially if resources are accessed concurrently by different components of your application.

- Resource update operations may not complete immediately. For example, [updating an index](#) or an [indexer](#) may take several seconds to complete. Resource updates are *serialized*, which means multiple update operations may not run simultaneously on the same resource.
- When two clients concurrently update a resource without coordination, a *race condition* is possible. One client could start an update operation while the other client receives a conflict error. To prevent this, Azure AI Search supports an *optimistic concurrency model*. There are no locks on a resource. Instead, there's an ETag for every resource that identifies the resource version so that you can formulate requests that avoid accidental overwrites.

How it works

Optimistic concurrency is implemented through access condition checks in API calls writing to indexes, indexers, data sources, skillsets, knowledge agents, and synonymMap resources.

All resources have an [entity tag \(ETag\)](#) that provides object version information. By checking the ETag first, you can avoid concurrent updates in a typical workflow (get, modify locally, update) by ensuring the resource's ETag matches your local copy.

- The REST API uses an [ETag](#) on the request header.
- The Azure SDK for .NET sets the ETag through an `accessCondition` class, setting the [If-Match | If-Match-None header](#) on the resource. Objects that use ETags, such as [SynonymMap.ETag](#) and [SearchIndex.ETag](#), have an `accessCondition` class.

Every time you update a resource, its ETag changes automatically. When you implement concurrency management, all you're doing is putting a precondition on the update request that requires the remote resource to have the same ETag as the copy of the resource that you modified on the client. If another process changes the remote resource, the ETag doesn't match the precondition and the request fails with HTTP 412 or 409. If you're using the .NET SDK, this failure manifests as an exception where the `IsAccessConditionFailed()` extension method returns true.

Note

There is only one mechanism for concurrency. It's always used regardless of which API or SDK is used for resource updates. Starting July 18, 2025, Azure AI Search began enforcing serialization for index creation and update operations to ensure consistency and reliability.

Example

The following code demonstrates optimistic concurrency for an update operation. It fails the second update because the object's ETag is changed by a previous update. More specifically, when the ETag in the request header no longer matches the ETag of the object, the search service return a status code of 400 (bad request), and the update fails.

C#

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using System;
using System.Net;
using System.Threading.Tasks;

namespace AzureSearch.SDKHowTo
{
    class Program
    {
        // This sample shows how ETags work by performing conditional updates and
        // deletes
        // on an Azure Search index.
        static void Main(string[] args)
        {
            string serviceName = "PLACEHOLDER FOR YOUR SEARCH SERVICE NAME";
            string apiKey = "PLACEHOLDER FOR YOUR SEARCH SERVICE ADMIN API KEY";

            // Create a SearchIndexClient to send create/delete index commands
            Uri serviceEndpoint = new
Uri($"https://'{serviceName}'.search.windows.net/");
            AzureKeyCredential credential = new AzureKeyCredential(apiKey);
            SearchIndexClient adminClient = new SearchIndexClient(serviceEndpoint,
            credential);

            // Delete index if it exists
            Console.WriteLine("Check for index and delete if it already
exists...\n");
            DeleteTestIndexIfExists(adminClient);

            // Every top-level resource in Azure Search has an associated ETag
            // that keeps track of which version
            // of the resource you're working on. When you first create a resource
            such as an index, its ETag is
            // empty.
```

```

    SearchIndex index = DefineTestIndex();

    Console.WriteLine(
        $"Test searchIndex hasn't been created yet, so its ETag should be
blank. ETag: '{index.ETag}'");

        // Once the resource exists in Azure Search, its ETag is populated.
Make sure to use the object
        // returned by the SearchIndexClient. Otherwise, you will still have
the old object with the
        // blank ETag.
        Console.WriteLine("Creating index...\n");
        index = adminClient.CreateIndex(index);
        Console.WriteLine($"Test index created; Its ETag should be populated.
ETag: '{index.ETag}'");

        // ETags prevent concurrent updates to the same resource. If another
        // client tries to update the resource, it will fail as long as all
clients are using the right
        // access conditions.
        SearchIndex indexForClientA = index;
        SearchIndex indexForClientB = adminClient.GetIndex("test-idx");

        Console.WriteLine("Simulating concurrent update. To start, clients A
and B see the same ETag.");
        Console.WriteLine($"ClientA ETag: '{indexForClientA.ETag}' ClientB
ETag: '{indexForClientB.ETag}'");

        // indexForClientA successfully updates the index.
        indexForClientA.Fields.Add(new SearchField("a",
SearchFieldDataType.Int32));
        indexForClientA = adminClient.CreateOrUpdateIndex(indexForClientA);

        Console.WriteLine($"Client A updates test-idx by adding a new field.
The new ETag for test-idx is: '{indexForClientA.ETag}'");

        // indexForClientB tries to update the index, but fails due to the
ETag check.
        try
        {
            indexForClientB.Fields.Add(new SearchField("b",
SearchFieldDataType.Boolean));
            adminClient.CreateOrUpdateIndex(indexForClientB);

            Console.WriteLine("Whoops; This shouldn't happen");
            Environment.Exit(1);
        }
        catch (RequestFailedException e) when (e.Status == 400)
        {
            Console.WriteLine("Client B failed to update the index, as
expected.");
        }

        // Uncomment the next line to remove test-idx

```

```

        //adminClient.DeleteIndex("test-idx");
        Console.WriteLine("Complete. Press any key to end application...\n");
        Console.ReadKey();
    }

private static void DeleteTestIndexIfExists(SearchIndexClient adminClient)
{
    try
    {
        if (adminClient.GetIndex("test-idx") != null)
        {
            adminClient.DeleteIndex("test-idx");
        }
    }
    catch (RequestFailedException e) when (e.Status == 404)
    {
        //if an exception occurred and status is "Not Found", this is
        working as expected
        Console.WriteLine("Failed to find index and this is because it's
not there.");
    }
}

private static SearchIndex DefineTestIndex() =>
    new SearchIndex("test-idx", new[] { new SearchField("id",
SearchFieldDataType.String) { IsKey = true } });
}

```

Design pattern

A design pattern for implementing optimistic concurrency should include a loop that retries the access condition check, a test for the access condition, and optionally retrieves an updated resource before attempting to reapply the changes.

This code snippet illustrates the addition of a synonymMap to an index that already exists.

The snippet gets the hotels-sample-index index, checks the object version on an update operation, throws an exception if the condition fails, and then retries the operation (up to three times), starting with index retrieval from the server to get the latest version.

C#

```

private static void EnableSynonymsInHotelsIndexSafely(SearchIndexClient
indexClient)
{
    int MaxNumTries = 3;

    for (int i = 0; i < MaxNumTries; ++i)

```

```

{
    try
    {
        SearchIndex index = indexClient.GetIndex("hotels-sample-index");
        index = AddSynonymMapsToFields(index);

        // The onlyIfUnchanged condition ensures that the index is updated only
        // if the ETags match.
        indexClient.CreateOrUpdateIndex(index, onlyIfUnchanged: true);

        Console.WriteLine("Updated the index successfully.\n");
        break;
    }
    catch (RequestFailedException e) when (e.Status == 412)
    {
        Console.WriteLine($"Index update failed : {e.Message}.

Attempt{i}/{MaxNumTries}.\n");
    }
}

private static SearchIndex AddSynonymMapsToFields(SearchIndex index)
{
    index.Fields.First(f => f.Name == "category").SynonymMapNames.Add("desc-
synonymmap");
    index.Fields.First(f => f.Name == "tags").SynonymMapNames.Add("desc-
synonymmap");
    return index;
}

```

See also

- [ETag Struct](#)
- [SearchIndex.ETag Property](#)
- [SearchIndexClient.CreateOrUpdateIndex Method](#)
- [HTTP Status Codes](#)

Monitor Azure AI Search

07/25/2025

This article describes:

- The types of monitoring data you can collect for this service.
- Ways to analyze that data.

 **Note**

If you're already familiar with this service and/or Azure Monitor and just want to know how to analyze monitoring data, see the [Analyze](#) section near the end of this article.

When you have critical applications and business processes that rely on Azure resources, you need to monitor and get alerts for your system. The Azure Monitor service collects and aggregates metrics and logs from every component of your system. Azure Monitor provides you with a view of availability, performance, and resilience, and notifies you of issues. You can use the Azure portal, PowerShell, Azure CLI, REST API, or client libraries to set up and view monitoring data.

- For more information on Azure Monitor, see the [Azure Monitor overview](#).
- For more information on how to monitor Azure resources in general, see [Monitor Azure resources with Azure Monitor](#).

 **Note**

Azure AI Search doesn't log the identity of the person or app accessing content or operations on the search service. If you require this level of monitoring, you need to implement it in your client application.

Resource types

Azure uses the concept of resource types and IDs to identify everything in a subscription. Resource types are also part of the resource IDs for every resource running in Azure. For example, one resource type for a virtual machine is `Microsoft.Compute/virtualMachines`. For a list of services and their associated resource types, see [Resource providers](#).

Azure Monitor similarly organizes core monitoring data into metrics and logs based on resource types, also called *namespaces*. Different metrics and logs are available for different resource types. Your service might be associated with more than one resource type.

For more information about the resource types for Azure AI Search, see [Azure AI Search monitoring data reference](#).

Data storage

For Azure Monitor:

- Metrics data is stored in the Azure Monitor metrics database.
- Log data is stored in the Azure Monitor logs store. Log Analytics is a tool in the Azure portal that can query this store.
- The Azure activity log is a separate store with its own interface in the Azure portal.

You can optionally route metric and activity log data to the Azure Monitor logs store. You can then use Log Analytics to query the data and correlate it with other log data.

Many services can use diagnostic settings to send metric and log data to other storage locations outside Azure Monitor. Examples include Azure Storage, [hosted partner systems](#), and [non-Azure partner systems](#), by using Event Hubs.

For detailed information on how Azure Monitor stores data, see [Azure Monitor data platform](#).

Azure Monitor platform metrics

Azure Monitor provides platform metrics for most services. These metrics are:

- Individually defined for each namespace.
- Stored in the Azure Monitor time-series metrics database.
- Lightweight and capable of supporting near real-time alerting.
- Used to track the performance of a resource over time.

Collection: Azure Monitor collects platform metrics automatically. No configuration is required.

Routing: You can also route some platform metrics to Azure Monitor Logs / Log Analytics so you can query them with other log data. Check the **DS export** setting for each metric to see if you can use a diagnostic setting to route the metric to Azure Monitor Logs / Log Analytics.

- For more information, see the [Metrics diagnostic setting](#).
- To configure diagnostic settings for a service, see [Create diagnostic settings in Azure Monitor](#).

For a list of all metrics it's possible to gather for all resources in Azure Monitor, see [Supported metrics in Azure Monitor](#).

In Azure AI Search, platform metrics measure query performance, indexing volume, and skillset invocation. For a list of available metrics for Azure AI Search, see [Azure AI Search monitoring data reference](#).

To learn how to analyze query and index performance, see [Analyze performance in Azure AI Search](#).

Azure Monitor resource logs

Resource logs provide insight into operations that were done by an Azure resource. Logs are generated automatically, but you must route them to Azure Monitor logs to save or query them. Logs are organized in categories. A given namespace might have multiple resource log categories.

Collection: Resource logs aren't collected and stored until you create a *diagnostic setting* and route the logs to one or more locations. When you create a diagnostic setting, you specify which categories of logs to collect. There are multiple ways to create and maintain diagnostic settings, including the Azure portal, programmatically, and through Azure Policy.

Routing: The suggested default is to route resource logs to Azure Monitor Logs so you can query them with other log data. Other locations such as Azure Storage, Azure Event Hubs, and certain Microsoft monitoring partners are also available. For more information, see [Azure resource logs](#) and [Resource log destinations](#).

For detailed information about collecting, storing, and routing resource logs, see [Diagnostic settings in Azure Monitor](#).

For a list of all available resource log categories in Azure Monitor, see [Supported resource logs in Azure Monitor](#).

All resource logs in Azure Monitor have the same header fields, followed by service-specific fields. The common schema is outlined in [Azure Monitor resource log schema](#).

For the available resource log categories, their associated Log Analytics tables, and the logs schemas for Azure AI Search, see [Azure AI Search monitoring data reference](#).

Azure activity log

The activity log contains subscription-level events that track operations for each Azure resource as seen from outside that resource; for example, creating a new resource or starting a virtual machine.

Collection: Activity log events are automatically generated and collected in a separate store for viewing in the Azure portal.

Routing: You can send activity log data to Azure Monitor Logs so you can analyze it alongside other log data. Other locations such as Azure Storage, Azure Event Hubs, and certain Microsoft monitoring partners are also available. For more information on how to route the activity log, see [Overview of the Azure activity log](#).

In Azure AI Search, activity logs reflect control plane activity such as service creation and configuration, or API key usage or management. Entries often include **Get Admin Key**, one entry for every call that [provided an admin API key](#) on the request. There are no details about the call itself, just a notification that the admin key was used.

API keys can be disabled for data plane operations, such as creating or querying an index, but on the control plane they're used in the Azure portal to return service information. Control plane operations can request API keys so you continue to see key-related requests in the Activity log even if you disable key-based authentication.

The following screenshot shows Azure AI Search activity log signals you can configure in an alert.

The screenshot shows a configuration interface for setting up an alert. At the top, there's a note: "Choose a signal below and configure the logic on the next screen to define the alert condition." Below this, there are two dropdown menus: "Signal type" set to "Activity Log" and "Monitor service" set to "All". A message indicates "Displaying 1 - 9 signals out of total 9 signals". A search bar labeled "Search by signal name" is present. A table lists nine signals, each with a small icon, the signal name, its type, and the monitor service it belongs to. The signals listed are: All Administrative operations, Set Search Service (Microsoft.Search/searchServices), Delete Search Service (Microsoft.Search/searchServices), Start Search Service (Microsoft.Search/searchServices), Stop Search Service (Microsoft.Search/searchServices), Get Admin Key (Microsoft.Search/searchServices), Regenerate Admin Key (Microsoft.Search/searchServices), Get Query Keys (Microsoft.Search/searchServices), and Create Query Key (Microsoft.Search/searchServices). All signals are categorized under the "Activity log" type and belong to the "Administrative" monitor service.

Signal name	Signal type	Monitor service
All Administrative operations	Activity log	Administrative
Set Search Service (Microsoft.Search/searchServices)	Activity log	Administrative
Delete Search Service (Microsoft.Search/searchServices)	Activity log	Administrative
Start Search Service (Microsoft.Search/searchServices)	Activity log	Administrative
Stop Search Service (Microsoft.Search/searchServices)	Activity log	Administrative
Get Admin Key (Microsoft.Search/searchServices)	Activity log	Administrative
Regenerate Admin Key (Microsoft.Search/searchServices)	Activity log	Administrative
Get Query Keys (Microsoft.Search/searchServices)	Activity log	Administrative
Create Query Key (Microsoft.Search/searchServices)	Activity log	Administrative

For other entries, see the [Management REST API reference](#) for control plane activity that might appear in the log.

Analyze monitoring data

There are many tools for analyzing monitoring data.

Azure Monitor tools

Azure Monitor supports the following basic tools:

- [Metrics explorer](#), a tool in the Azure portal that allows you to view and analyze metrics for Azure resources. For more information, see [Analyze metrics with Azure Monitor metrics explorer](#).
- [Log Analytics](#), a tool in the Azure portal that allows you to query and analyze log data by using the [Kusto query language \(KQL\)](#). For more information, see [Get started with log queries in Azure Monitor](#).
- The [activity log](#), which has a user interface in the Azure portal for viewing and basic searches. To do more in-depth analysis, you have to route the data to Azure Monitor logs and run more complex queries in Log Analytics.

Tools that allow more complex visualization include:

- [Dashboards](#) that let you combine different kinds of data into a single pane in the Azure portal.
- [Workbooks](#), customizable reports that you can create in the Azure portal. Workbooks can include text, metrics, and log queries.
- [Grafana](#), an open platform tool that excels in operational dashboards. You can use Grafana to create dashboards that include data from multiple sources other than Azure Monitor.
- [Power BI](#), a business analytics service that provides interactive visualizations across various data sources. You can configure Power BI to automatically import log data from Azure Monitor to take advantage of these visualizations.

Azure Monitor export tools

You can get data out of Azure Monitor into other tools by using the following methods:

- **Metrics:** Use the [REST API for metrics](#) to extract metric data from the Azure Monitor metrics database. The API supports filter expressions to refine the data retrieved. For more information, see [Azure Monitor REST API reference](#).
- **Logs:** Use the REST API or the [associated client libraries](#).
- Another option is the [workspace data export](#).

To get started with the REST API for Azure Monitor, see [Azure monitoring REST API walkthrough](#).

Kusto queries

You can analyze monitoring data in the Azure Monitor Logs / Log Analytics store by using the Kusto query language (KQL).

Important

When you select **Logs** from the service's menu in the portal, Log Analytics opens with the query scope set to the current service. This scope means that log queries will only include data from that type of resource. If you want to run a query that includes data from other Azure services, select **Logs** from the **Azure Monitor** menu. See [Log query scope and time range in Azure Monitor Log Analytics](#) for details.

For a list of common queries for any service, see the [Log Analytics queries interface](#).

The following queries can get you started. See [Analyze performance in Azure AI Search](#) for more examples and guidance specific to search service.

List metrics by name

Return a list of metrics and the associated aggregation. The query is scoped to the current search service over the time range that you specify.

```
Kusto
```

```
AzureMetrics  
| project MetricName, Total, Count, Maximum, Minimum, Average
```

List operations by name

Return a list of operations and a count of each one.

```
Kusto
```

```
AzureDiagnostics  
| summarize count() by OperationName
```

Long-running queries

This Kusto query against AzureDiagnostics returns `Query.Search` operations, sorted by duration (in milliseconds). For more examples of `Query.Search` queries, see [Analyze performance in Azure AI Search](#).

Kusto

```
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d,
IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

Indexer status

This Kusto query returns the status of indexer operations. Results include the operation name, description of the request (which includes the name of the indexer), result status (Success or Failure), and the [HTTP status code](#). For more information about indexer execution, see [Monitor indexer status](#).

Kusto

```
AzureDiagnostics
| project OperationName, Description_s, Documents_d, ResultType, resultSignature_d
| where OperationName == "Indexers.Status"
```

Alerts

Azure Monitor alerts proactively notify you when specific conditions are found in your monitoring data. Alerts allow you to identify and address issues in your system before your customers notice them. For more information, see [Azure Monitor alerts](#).

There are many sources of common alerts for Azure resources. For examples of common alerts for Azure resources, see [Sample log alert queries](#). The [Azure Monitor Baseline Alerts \(AMBA\)](#) site provides a semi-automated method of implementing important platform metric alerts, dashboards, and guidelines. The site applies to a continually expanding subset of Azure services, including all services that are part of the Azure Landing Zone (ALZ).

The common alert schema standardizes the consumption of Azure Monitor alert notifications. For more information, see [Common alert schema](#).

Types of alerts

You can alert on any metric or log data source in the Azure Monitor data platform. There are many different types of alerts depending on the services you're monitoring and the monitoring data you're collecting. Different types of alerts have various benefits and drawbacks. For more information, see [Choose the right monitoring alert type](#).

The following list describes the types of Azure Monitor alerts you can create:

- [Metric alerts](#) evaluate resource metrics at regular intervals. Metrics can be platform metrics, custom metrics, logs from Azure Monitor converted to metrics, or Application Insights metrics. Metric alerts can also apply multiple conditions and dynamic thresholds.
- [Log alerts](#) allow users to use a Log Analytics query to evaluate resource logs at a predefined frequency.
- [Activity log alerts](#) trigger when a new activity log event occurs that matches defined conditions. Resource Health alerts and Service Health alerts are activity log alerts that report on your service and resource health.

Some Azure services also support [smart detection alerts](#), [Prometheus alerts](#), or [recommended alert rules](#).

For some services, you can monitor at scale by applying the same metric alert rule to multiple resources of the same type that exist in the same Azure region. Individual notifications are sent for each monitored resource. For supported Azure services and clouds, see [Monitor multiple resources with one alert rule](#).

Azure AI Search alert rules

The following table lists common and recommended alert rules for Azure AI Search. On a search service, throttling or query latency that exceeds a given threshold are the most commonly used alerts, but you might also want to be notified if a search service is deleted.

[] [Expand table](#)

Alert type	Condition	Description
Search Latency (metric alert)	Whenever the average search latency is greater than a user-specified threshold (in seconds)	Send an SMS alert when average query response time exceeds the threshold.
Throttled search queries percentage (metric alert)	Whenever the total throttled search queries percentage is greater than or equal to a user-specified threshold	Send an SMS alert when dropped queries begin to exceed the threshold.

Alert type	Condition	Description
Delete Search Service (activity log alert)	Whenever the Activity Log has an event with Category='Administrative', Signal name='Delete Search Service (searchServices)', Level='critical'	Send an email if a search service is deleted in the subscription.

Advisor recommendations

For some services, if critical conditions or imminent changes occur during resource operations, an alert displays on the service **Overview** page in the portal. You can find more information and recommended fixes for the alert in **Advisor recommendations** under **Monitoring** in the left menu. During normal operations, no advisor recommendations display.

For more information on Azure Advisor, see [Azure Advisor overview](#).

Related content

- [Azure AI Search monitoring data reference](#)
- [Monitor Azure resources with Azure Monitor](#)
- [Monitor queries](#)
- [Monitor indexer-based indexing](#)
- [Visualize resource logs](#)
- [Analyze performance in Azure AI Search](#)
- [Tips for better performance](#)

Configure diagnostic logging for Azure AI Search

08/08/2025

Diagnostic logs provide insight into operations that occur in your Azure AI Search resource. In contrast to Activity Logs that track operations performed on Azure resources at the subscription level, known as the [control plane](#), diagnostic logging monitors operations on the search service itself. Diagnostic logging is essential for effective oversight of service operations like indexing and queries.

This article explains how to enable diagnostic logging and find information about system and user operations on an Azure AI Search resource.

ⓘ Note

Azure AI Search doesn't log the identity of the person or app accessing content or operations on the search service. If you require this level of monitoring, you need to implement it in your client application.

Prerequisites

- An [Azure Log Analytics workspace](#) in the same subscription.

Enable diagnostic logging

1. Sign in to the [Azure portal](#) and [find your search service](#).
2. Under **Monitoring > Diagnostic settings**, select **Add diagnostic setting**.
3. Provide a descriptive name that identifies the service and level of logging, such as "my-search-service-all-logs" or "my-search-service-audit-logs".
4. Under **Logs**, choose a category:
 - **Audit logs** capture user or app interactions with data or the settings of the service, but don't include user or groups identities.
 - **Operation logs** capture information about operations on a search service.
 - **allLogs** collect everything.

Verbose logging can be expensive to store and complex to manage and store. You might want to start with **allLogs** and then switch to more scoped logging if it meets your information requirements. For more information about these categories, see [Diagnostic settings in Azure Monitor](#).

5. For a destination, we recommend **Send to Log Analytics workspace** so that you can run Kusto queries against the data. Provide an existing Log Analytics workspace to store your logs.
6. Save the settings.

Repeat these steps if you require a more [comprehensive data collection strategy](#).

Each diagnostic setting you create requires separate storage. If you use the Azure portal to review logs, the first diagnostic setting is used by default. You can navigate to specific workspaces for visualization support.

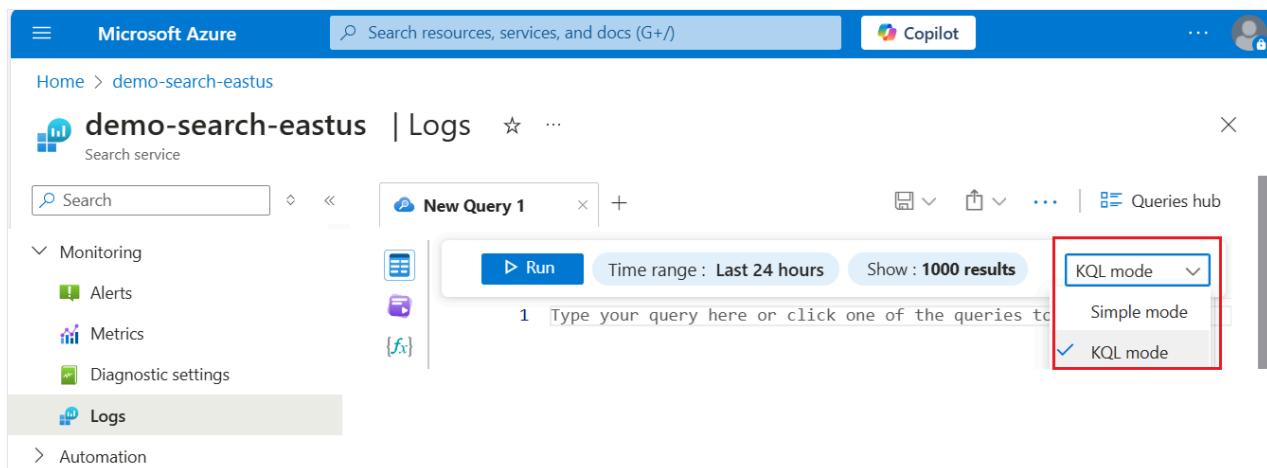
Note

If you're using [key-based authentication](#), Azure AI Search can't monitor individual user access to content on the search service. If you require this level of monitoring, you need to implement it in your client application.

View logs in Log Analytics

Follow these instructions to explore log analytics data for your search service.

1. Under **Monitoring**, select **Logs**. Query hub opens by default. You can try the available queries, or close the hub and open a query window in KQL mode to run queries written in the [Kusto Query Language \(KQL\)](#).



2. In a query window, you can run Kusto queries against your logs.

The screenshot shows the Azure Kusto Query Editor interface. At the top, there's a header bar with a 'New Query 1*' tab, a '+' button, and several icons for saving, opening, and queries hub. Below the header is a toolbar with a 'Run' button, a time range selector set to 'Last 24 hours', a result count selector set to '1000 results', and a 'KQL mode' dropdown. On the left, there are icons for file operations, a function editor, and a refresh. The main area contains a code editor with the following KQL query:

```
1 AzureDiagnostics
2 | where ResourceProvider == "MICROSOFT.SEARCH"
3 | summarize avg(DurationMs)
4 by OperationName
```

Below the code editor is a results table with two columns: 'OperationName' and 'avg_DurationMs'. The data is as follows:

OperationName	avg_DurationMs
> ServiceStats	19.285714285714285
> Indexes.ListIndexStatsSummaries	30.428571428571427
> Indexing.Index	1908.857142857143
> Query.Search	98.2
> Indexes.Get	25.333333333333332
> Indexers.Status	23
> Indexers.List	22
> CORS.Preflight	19.333333333333332

Sample Kusto queries

Here are a few basic Kusto queries you can use to explore your log data.

Run this query for all diagnostic logs from Azure AI Search services over the specified time period:

```
Kusto

AzureDiagnostics
| where ResourceProvider == "MICROSOFT.SEARCH"
```

Run this query to see the 10 most recent logs:

```
Kusto

AzureDiagnostics
| where ResourceProvider == "MICROSOFT.SEARCH"
| take 10
```

Run this query to group operations by Resource:

```
Kusto
```

```
AzureDiagnostics  
| where ResourceProvider == "MICROSOFT.SEARCH" |  
summarize count() by Resource
```

Run this query to find the average time it takes to perform an operation:

Kusto

```
AzureDiagnostics  
| where ResourceProvider == "MICROSOFT.SEARCH"  
| summarize avg(DurationMs)  
by OperationName
```

Run this query to view the volume of operations over time split by OperationName with counts binned for every 10 seconds.

Kusto

```
AzureDiagnostics  
| where ResourceProvider == "MICROSOFT.SEARCH"  
| summarize count()  
by bin(TimeGenerated, 10s), OperationName  
| render areachart kind=unstacked
```

Monitor query requests in Azure AI Search

08/08/2025

This article explains how to measure query performance and volume using built-in metrics and diagnostic logging. It also explains how to get the query strings entered by application users.

The Azure portal shows basic metrics about query latency, query load (QPS), and throttling. Historical data that feeds into these metrics can be accessed in the Azure portal for 30 days. For longer retention, or to report on operational data and query strings, you must [enable diagnostic logging](#) and choose a storage option for persisting logged operations and metrics. We recommend **Log Analytics workspace** as a destination for logged operations. Kusto queries and data exploration target a Log Analytics workspace.

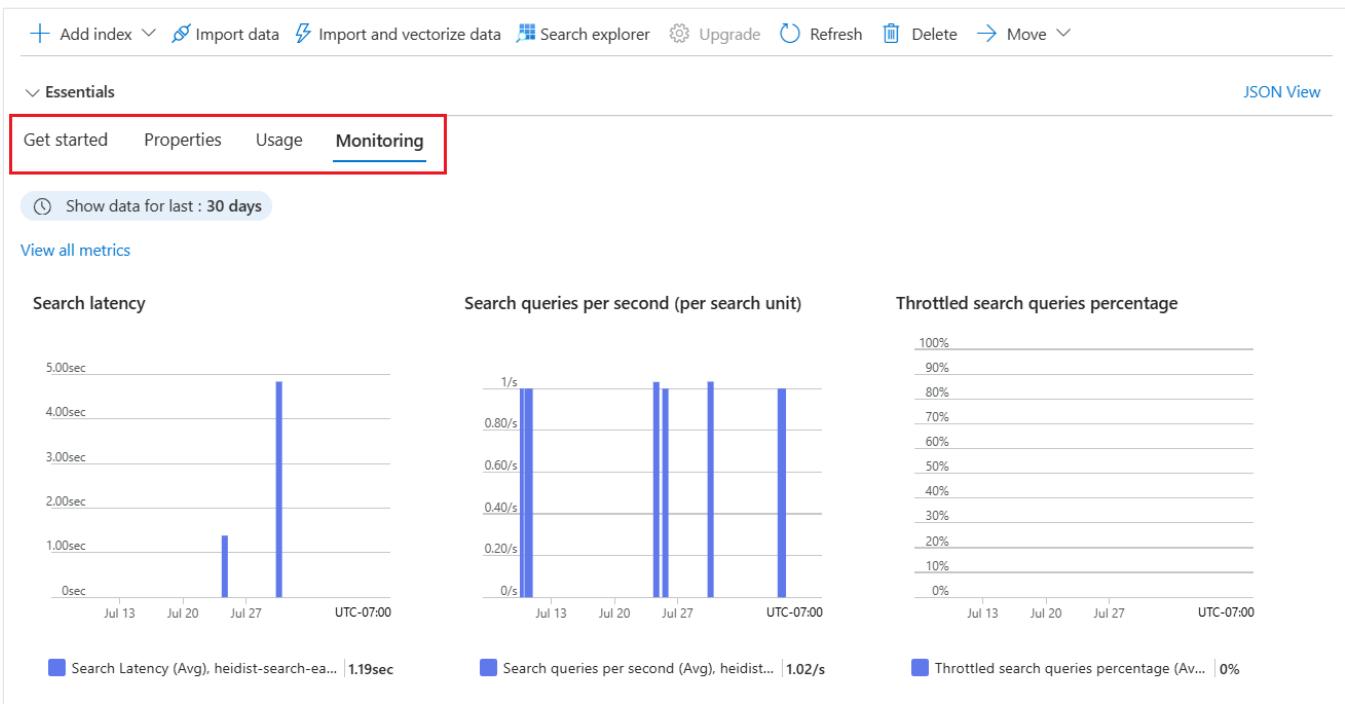
Conditions that maximize the integrity of data measurement include:

- Use a billable service (a service created at either the Basic or a Standard tier). The free service is shared by multiple subscribers, which introduces a certain amount of volatility as loads shift.
- Use a single replica and partition, if possible, to create a contained and isolated environment. If you use multiple replicas, query metrics are averaged across multiple nodes, which can lower the precision of results. Similarly, multiple partitions mean that data is divided, with the potential that some partitions might have different data if indexing is also underway. When you tune query performance, a single node and partition gives a more stable environment for testing.

Query volume (QPS)

Volume is measured as **Search Queries Per Second** (QPS), a built-in metric that can be reported as an average, count, minimum, or maximum values of queries that execute within a one-minute window. One-minute intervals (`TimeGrain = "PT1M"`) for metrics is fixed within the system.

Azure AI Search retains 30-days of metrics data by default. You can enable logging for longer retention. QPS is available in the Azure portal, in the **Monitoring** tab of your search service.



To learn more about the `SearchQueriesPerSecond` metric, see [Search queries per second](#).

Query performance

Service-wide, query performance is measured as *search latency* and *throttled queries*. These metrics are also available on the **Monitoring** tab.

Search latency

Search latency indicates how long a query takes to complete. To learn more about the `SearchLatency` metric, see [Search latency](#).

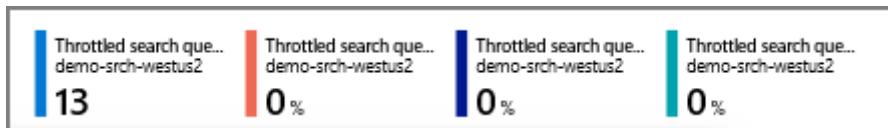
Consider the following example of **Search Latency** metrics: 86 queries were sampled, with an average duration of 23.26 milliseconds. A minimum of 0 indicates some queries were dropped. The longest running query took 1000 milliseconds to complete. Total execution time was 2 seconds.

Search Latency (Avg) demo-srch-westus2 23.26 ms	Search Latency (Min) demo-srch-westus2 0 sec	Search Latency (Max) demo-srch-westus2 1000.00 ms	Search Latency (Sum) demo-srch-westus2 2.00 sec	Search Latency (Count) demo-srch-westus2 86
--	---	--	--	--

Throttled queries

Throttled queries refers to queries that are dropped instead of processed. In most cases, throttling is a normal part of running the service. It isn't necessarily an indication that there's something wrong. To learn more about the `ThrottledSearchQueriesPercentage` metric, see [Throttled search queries percentage](#).

In the following screenshot, the first number is the count (or number of metrics sent to the log). Other aggregations, which appear at the top or when hovering over the metric, include average, maximum, and total. In this sample, no requests were dropped.

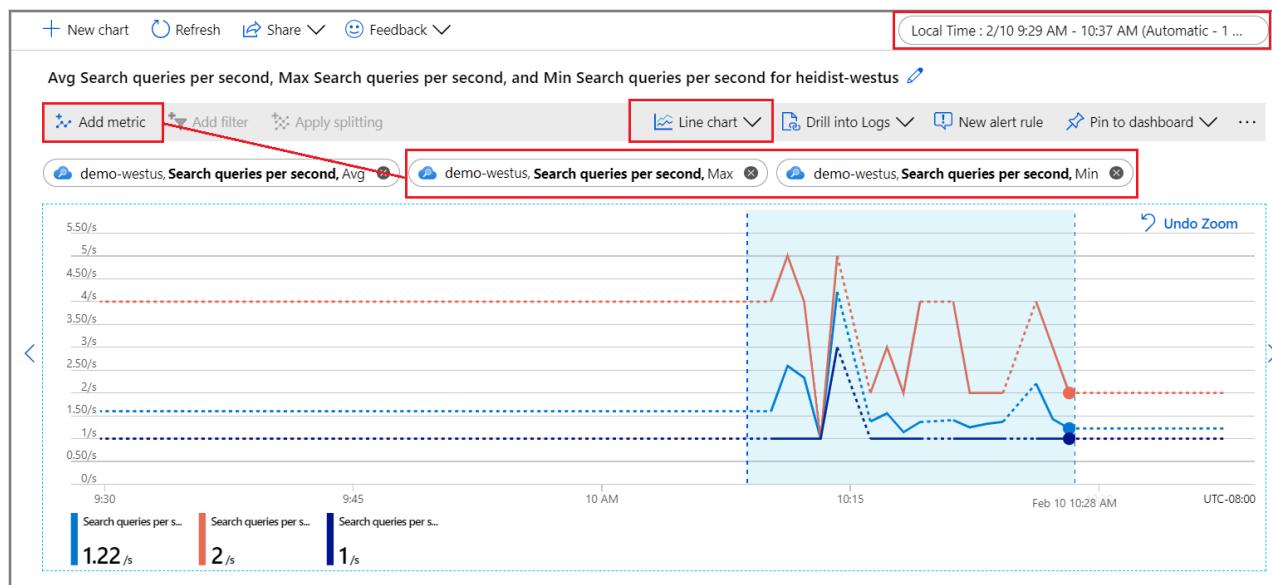


Explore metrics in the Azure portal

For a quick look at the current numbers, the **Monitoring** tab on the service Overview page shows three metrics (**Search latency**, **Search queries per second (per search unit)**, **Throttled Search Queries Percentage**) over fixed intervals measured in hours, days, and weeks, with the option of changing the aggregation type.

For deeper exploration, open metrics explorer from the **Monitoring** menu so that you can layer, zoom in, and visualize data to explore trends or anomalies. Learn more about metrics explorer by completing this [tutorial on creating a metrics chart](#).

1. Under the Monitoring section, select **Metrics** to open the metrics explorer with the scope set to your search service.
2. Under Metric, choose one from the dropdown list and review the list of available aggregations for a preferred type. The aggregation defines how the collected values are sampled over each time interval.



3. In the top-right corner, set the time interval.
4. Choose a visualization. The default is a line chart.
5. Layer more aggregations by choosing **Add metric** and selecting different aggregations.

6. Zoom into an area of interest on the line chart. Put the mouse pointer at the beginning of the area, select and hold the left mouse button, drag to the other side of area, and release the button. The chart zooms in on that time range.

Return query strings entered by users

When you enable resource logging, the system captures query requests in the **AzureDiagnostics** table. As a prerequisite, you must have already specified a destination for [logged operations](#), either a log analytics workspace or another storage option.

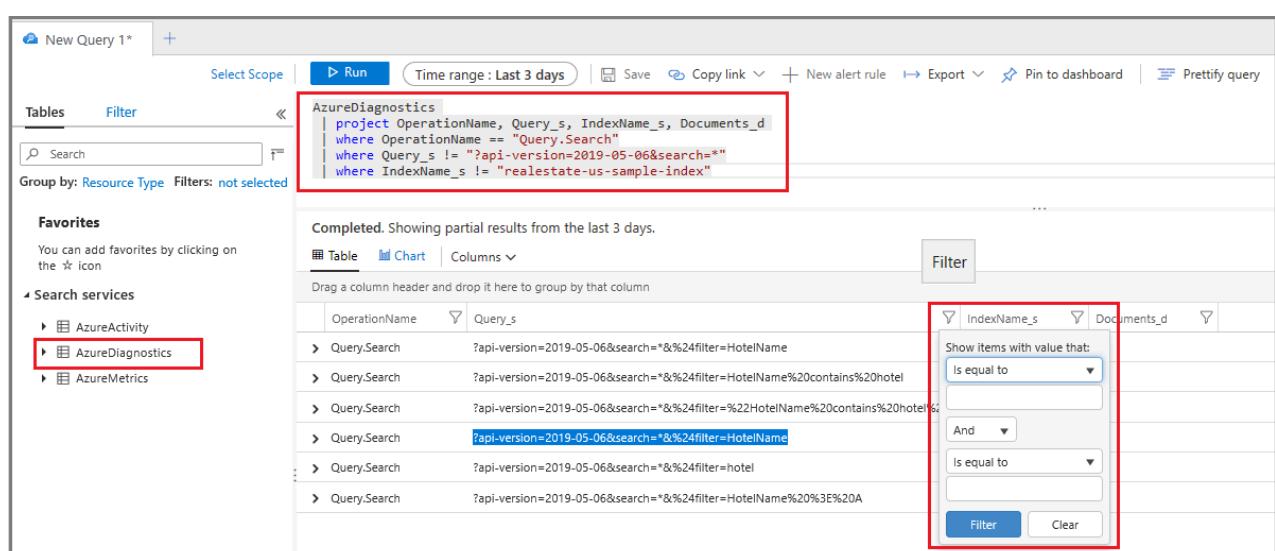
1. Under the Monitoring section, select **Logs** to open up an empty query window in Log Analytics.
2. Run the following expression to search `Query.Search` operations, returning a tabular result set consisting of the operation name, query string, the index queried, and the number of documents found. The last two statements exclude query strings consisting of an empty or unspecified search, over a sample index, which cuts down the noise in your results.



```
Kusto

AzureDiagnostics
| project OperationName, Query_s, IndexName_s, Documents_d
| where OperationName == "Query.Search"
| where Query_s != "?api-version=2024-07-01&search=*&filter=HotelName"
| where IndexName_s != "hotels-sample-index"
```

3. Optionally, set a Column filter on `Query_s` to search over a specific syntax or string. For example, you could filter over `is equal to ?api-version=2024-07-01&search=*&%24filter=HotelName`.



The screenshot shows the Azure Log Analytics workspace with the Kusto Query Editor open. The query in the editor is:

```
AzureDiagnostics
| project OperationName, Query_s, IndexName_s, Documents_d
| where OperationName == "Query.Search"
| where Query_s != "?api-version=2019-05-06&search=*&%24filter=HotelName"
| where IndexName_s != "realestate-us-sample-index"
```

The 'Tables' pane on the left has 'AzureDiagnostics' selected. The 'Filter' pane on the right shows a 'Filter' dialog for the 'IndexName_s' column. The 'Show items with value that:' dropdown is set to 'Is equal to'. Below it, there is a text input field and a 'Filter' button. The 'And' dropdown is set to 'Is equal to' with a second input field and a 'Filter' button.

While this technique works for ad hoc investigation, building a report lets you consolidate and present the query strings in a layout more conducive to analysis.

Identify long-running queries

Add the duration column to get the numbers for all queries, not just those that are picked up as a metric. Sorting this data shows you which queries take the longest to complete.

1. Under the Monitoring section, select **Logs** to query for log information.
2. Run the following basic query to return queries, sorted by duration in milliseconds. The longest-running queries are at the top.

```
Kusto

AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d,
IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

The screenshot shows the Azure Kusto Query Editor interface. At the top, there's a code editor with the following query:

```
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d, IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

Below the code editor, the results pane displays the query results. It shows a table with the following columns: OperationName, result..., DurationMs, Query_s, Do..., and IndexName_s. The results are as follows:

OperationName	result...	DurationMs	Query_s	Do...	IndexName_s
Query.Search	200	636	?api-version=2019-05-06&search=restaurant&%24count=true&%24select=HotelId...	1	hotels-quickstart
Query.Search	200	357	?api-version=2019-05-06&search=*&%24count=true	14	demoindex
Query.Search	200	181	?api-version=2019-05-06&search=*	14	demoindex
Query.Search	200	154	?api-version=2019-05-06-Preview&search=*&%24count=true	14	demoindex
Query.Search	200	148	?api-version=2019-05-06&search=*&%24count=true&%24select=HotelId%2C%	4	hotels-quickstart
Query.Search	200	139	?api-version=2019-05-06&search=*	19	hotel-reviews-idx

Create a metric alert

A [metric alert](#) establishes a threshold for sending a notification or triggering a corrective action that you define in advance. You can create alerts related to query execution, but you can also create them for resource health, search service configuration changes, skill execution, and document processing (indexing).

All thresholds are user-defined, so you should have an idea of what activity level should trigger the alert.

For query monitoring, it's common to create a metric alert for search latency and throttled queries. If you know *when* queries are dropped, you can look for remedies that reduce load or increase capacity. For example, if throttled queries increase during indexing, you could postpone it until query activity subsides.

If you're pushing the limits of a particular replica-partition configuration, setting up alerts for query volume thresholds (QPS) is also helpful.

1. Under **Monitoring**, select **Alerts** and then select **Create alert rule**.
2. Under Condition, select **Add**.
3. Configure signal logic. For signal type, choose **metrics** and then select the signal.
4. After selecting the signal, you can use a chart to visualize historical data for an informed decision on how to proceed with setting up conditions.
5. Next, scroll down to Alert logic. For proof-of-concept, you could specify an artificially low value for testing purposes.
6. Next, specify or create an Action Group. This is the response to invoke when the threshold is met. It might be a push notification or an automated response.
7. Last, specify Alert details. Name and describe the alert, assign a severity value, and specify whether to create the rule in an enabled or disabled state.

If you specified an email notification, you receive an email from "Microsoft Azure" with a subject line of "Azure: Activated Severity: 3 <your rule name>".

Next steps

If you haven't done so already, review the fundamentals of search service monitoring to learn about the full range of oversight capabilities.

[Monitor operations and activity in Azure AI Search](#)

Monitor indexer status and results in Azure AI Search

08/08/2025

You can monitor indexer processing in the Azure portal, or programmatically through REST calls or an Azure SDK. In addition to status about the indexer itself, you can review start and end times, and detailed errors and warnings from a particular run.

Monitor using Azure portal

You can see the current status of all of your indexers in your search service portal page. Azure portal pages refresh every few minutes, so you don't see evidence of a new indexer run right away. Select Refresh at the top of the page to immediately retrieve the most recent view.

The screenshot shows the Azure portal interface for managing a search service. The top navigation bar includes 'Microsoft Azure', a search bar, and a 'Copilot' button. The main content area is titled 'demo-search-eastus | Indexers' and shows a table of indexers:

Status	Name	Last run	Docs succeeded	Errors/Warnings
Success	chunkingsample-indexer	5 months ago	0/0	0/0
Success	demoindexer	27 days ago	14/14	0/2
Success	cog-search-demo-idxr	1 day ago	14/14	0/5
Reset	vector-images-ai-vision-...	3 minutes ago	0/0	0/0
Failed	vector-nasa-ai-vision-m...	2 minutes ago	0/0	0/0
Success	hotels-sample-indexer	1 minute ago	50/50	0/0
In progress	doc-extraction-multimo...	44 seconds ago	0/0	0/0

The left sidebar lists various service components: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Search management (with Indexes selected), Data sources, Aliases, Skillsets, and Debug sessions. A 'Filter by name...' search bar is also present above the table.

[Expand table](#)

Status	Description
In Progress	Indicates active execution. the Azure portal reports on partial information. As indexing progresses, you can watch the Docs Succeeded value grow in response. Indexers that process large volumes of data can take a long time to run. For example, indexers that handle millions of source documents can run for 24 hours, and then restart almost immediately to pick up where it left off. As such, the status for high-volume indexers might always say In Progress in the Azure portal. Even when an indexer is running, details are available about ongoing progress and previous runs.

Status	Description
Success	Indicates the run was successful. An indexer run can be successful even if individual documents have errors, if the number of errors is less than the indexer's Max failed items setting.
Failed	The number of errors exceeded Max failed items and indexing stops.
Reset	The indexer's internal change tracking state was reset. The indexer runs in full, refreshing all documents, and not just those with newer timestamps.

You can select on an indexer in the list to see more details about the indexer's current and recent runs.

hotel-rooms-blob-indexer

Indexer

Run Reset Edit Delete

Indexer summary

Execution history

SUCCEEDED
FAILED

Execution details

LAST RESULT	LAST RUN	STATUS	DOCS SUCCEEDED
🔴	6/28, 03:29 UTC	Failed	1/2
🔵	6/28, 03:28 UTC	Reset	0/0
🟢	6/28, 00:00 UTC	Success	0/0
🟢	6/27, 22:02 UTC	Success	7/7
🔵	6/27, 22:02 UTC	Reset	0/0
🟢	6/27, 22:00 UTC	Success	7/7
🔵	6/27, 22:00 UTC	Reset	0/0
🟢	6/27, 21:23 UTC	Success	7/7
🔵	6/27, 21:23 UTC	Reset	0/0
🟢	6/27, 21:14 UTC	Success	7/7

1 2 3 4 5 < >

The **Indexer summary** chart displays a graph of the number of documents processed in its most recent runs.

The **Execution details** list shows up to 50 of the most recent execution results. Select on an execution result in the list to see specifics about that run. This includes its start and end times, and any errors and warnings that occurred.

The screenshot shows a modal window titled "myindexer" under the "Execution" tab. The main section is titled "Execution result" and displays the message "0 Errors/Warnings" with a green checkmark icon. Below this, a table lists execution details:

Status	Success
Datasource	test-cosmos-src
Target index	test-cosmosdb-index
Start	6/28, 03:27 UTC
End	6/28, 03:27 UTC
Documents succeeded	50
Documents failed	0

Below the table, there is a section titled "Errors" which contains an empty list [].

If there were document-specific problems during the run, they are listed in the Errors and Warnings fields.

hotel-rooms-blob-indexer

X

Execution

Execution result

1 Errors/Warnings !

Status	Failed
Datasource	sampleblobstore
Target index	hotel-rooms-sample
Start	6/28, 03:29 UTC
End	6/28, 03:29 UTC
Documents succeeded	0
Documents failed	1

Errors

```
[{"key": "https://sampleblobstore.blob.core.windows.net/hotel-rooms/Rooms11.json", "errorMessage": "Document key cannot be missing or empty.\r\n"}]
```

Warnings

```
[]
```

Warnings are common with some types of indexers, and don't always indicate a problem. For example indexers that use Azure AI services can report warnings when image or PDF files don't contain any text to process.

For more information about investigating indexer errors and warnings, see [Indexer troubleshooting guidance](#).

Monitor with Azure Monitoring Metrics

Azure AI Search is a monitored resource in Azure Monitor, which means that you can use [Metrics Explorer](#) to see basic metrics about the number of indexer-processed documents and

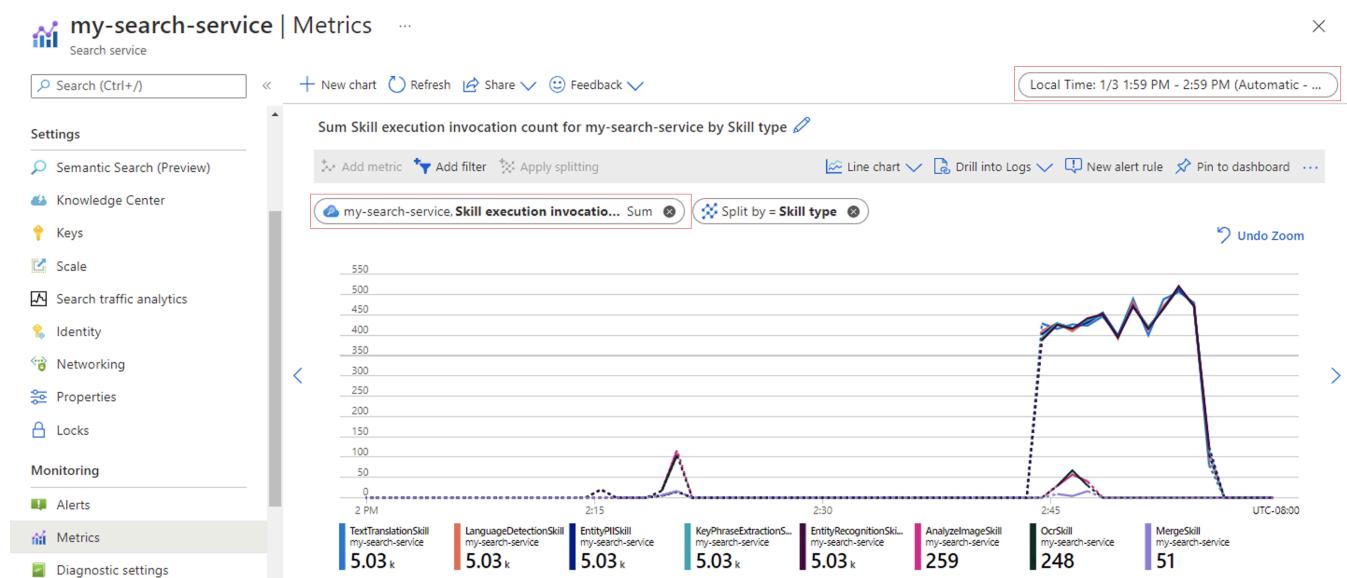
skill invocations. These metrics can be used to monitor indexer progress and [set up alerts](#).

Metric views can be filtered or split up by a set of predefined dimensions. To learn about the dimensions associated with the metrics *Document processed count* and *Skill execution invocation count*, see [Metric dimensions](#).

The following screenshot shows the number of documents processed by indexers within a service over an hour, split up by indexer name.



You can also configure the graph to see the number of skill invocations over the same hour interval.



Monitor using Get Indexer Status (REST API)

You can retrieve the status and execution history of an indexer using the [Get Indexer Status command](#):

HTTP

```
GET https://[service name].search.windows.net/indexers/[indexer name]/status?api-version=2024-07-01  
api-key: [Search service admin key]
```

The response contains overall indexer status, the last (or in-progress) indexer invocation, and the history of recent indexer invocations.

Output

```
{  
    "status": "running",  
    "lastResult": {  
        "status": "success",  
        "errorMessage": null,  
        "startTime": "2018-11-26T03:37:18.853Z",  
        "endTime": "2018-11-26T03:37:19.012Z",  
        "errors": [],  
        "itemsProcessed": 11,  
        "itemsFailed": 0,  
        "initialTrackingState": null,  
        "finalTrackingState": null  
    },  
    "executionHistory": [  
        {  
            "status": "success",  
            "errorMessage": null,  
            "startTime": "2018-11-26T03:37:18.853Z",  
            "endTime": "2018-11-26T03:37:19.012Z",  
            "errors": [],  
            "itemsProcessed": 11,  
            "itemsFailed": 0,  
            "initialTrackingState": null,  
            "finalTrackingState": null  
        }]  
}
```

Execution history contains up to the 50 most recent runs, which are sorted in reverse chronological order (most recent first).

Note there are two different status values. The top level status is for the indexer itself. An indexer status of **running** means the indexer is set up correctly and available to run, but not that it's currently running.

Each run of the indexer also has its own status that indicates whether that specific execution is ongoing (**running**), or already completed with a **success**, **transientFailure**, or **persistentFailure** status.

When an indexer is reset to refresh its change tracking state, a separate execution history entry is added with a **Reset** status.

For more information about status codes and indexer monitoring data, see [Get Indexer Status](#).

Monitor using .NET

The following C# example writes information about an indexer's status and the results of its most recent (or ongoing) run to the console.

C#

```
static void CheckIndexerStatus(SearchIndexerClient indexerClient, SearchIndexer
indexer)
{
    try
    {
        string indexerName = "hotels-sql-idxr";
        SearchIndexerStatus execInfo =
indexerClient.GetIndexerStatus(indexerName);

        Console.WriteLine("Indexer has run {0} times.",
execInfo.ExecutionHistory.Count);
        Console.WriteLine("Indexer Status: " + execInfo.Status.ToString());

        IndexerExecutionResult result = execInfo.LastResult;

        Console.WriteLine("Latest run");
        Console.WriteLine("Run Status: {0}", result.Status.ToString());
        Console.WriteLine("Total Documents: {0}, Failed: {1}", result.ItemCount,
result.FailedItemCount);

        TimeSpan elapsed = result.EndTime.Value - result.StartTime.Value;
        Console.WriteLine("StartTime: {0:T}, EndTime: {1:T}, Elapsed: {2:t}",
result.StartTime.Value, result.EndTime.Value, elapsed);

        string errorMsg = (result.ErrorMessage == null) ? "none" :
result.ErrorMessage;
        Console.WriteLine("ErrorMessage: {0}", errorMsg);
        Console.WriteLine(" Document Errors: {0}, Warnings: {1}\n",
result.Errors.Count, result.Warnings.Count);
    }
    catch (Exception e)
    {
        // Handle exception
    }
}
```

The output in the console looks something like this:

Output

```
Indexer has run 18 times.  
Indexer Status: Running  
Latest run  
  Run Status: Success  
  Total Documents: 7, Failed: 0  
  StartTime: 11:29:31 PM, EndTime: 11:29:31 PM, Elapsed: 00:00:00.2560000  
  ErrorMessage: none  
  Document Errors: 0, Warnings: 0
```

Note there are two different status values. The top-level status is the status of the indexer itself. An indexer status of **Running** means that the indexer is set up correctly and available for execution, but not that it's currently executing.

Each run of the indexer also has its own status for whether that specific execution is ongoing (**Running**), or was already completed with a **Success** or **TransientError** status.

When an indexer is reset to refresh its change tracking state, a separate history entry is added with a **Reset** status.

Next steps

For more information about status codes and indexer monitoring information, see the following API reference:

- [Indexers - Get Status \(REST API\)](#)
- [IndexerStatus](#)
- [IndexerExecutionStatus](#)
- [IndexerExecutionResult](#)

Visualize Azure AI Search Logs and Metrics with Power BI

05/29/2025

Azure AI Search can send operation logs and service metrics to an Azure Storage account, which can then be visualized in Power BI. This article explains the steps and how to use a Power BI template app to visualize the data. The template covers information about queries, indexing, operations, and service metrics.

(!) Note

The Power BI template currently uses a former product name, Azure Cognitive Search. The product name will be updated on the next template refresh.

Set up logging and install the template

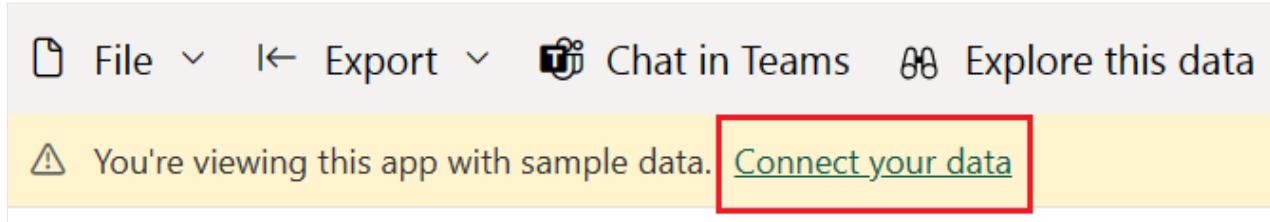
1. Enable metric and resource logging for your search service:
 - a. Create or identify an existing [Azure Storage account](#) where you can archive the logs.
 - b. Navigate to your search service in the Azure portal.
 - c. Under Monitoring, select **Diagnostic settings**.
 - d. Select **Add diagnostic setting**.
 - e. Check **Archive to a storage account**, provide your storage account information, and check **OperationLogs** and **AllMetrics**.
 - f. Select **Save**.
2. Once logging is enabled, logs and metrics are generated as you use the search service. It can take up to an hour before logged events show up in Azure Storage. Look for a **insights-logs-operationlogs** container for operations and a **insights-metrics-pt1m** container for metrics. Check your storage account for these containers to make sure you have data to visualize.
3. Find the Power BI app template in the [Power BI Apps marketplace](#) and install it into a new workspace or an existing workspace. The template is called **Azure Cognitive Search: Analyze Logs and Metrics**.
4. After installing the template, select it from your list of apps in Power BI.

Apps

Apps are collections of dashboards and reports in one easy-to-find place.

File	Name	Publisher	Published	App type	Version
	Azure Cognitive Search	4/24/2024 11:54:57 PM	4/24/24, 11:55:04 PM	Template app	Version 4

5. Select **Connect your data**.



6. Provide the name of the storage account that contains your logs and metrics. By default, the app looks at the last 10 days of data, but this value can be changed with the Days parameter.

X



Connect to Azure Cognitive Search

Get started setting up your app! Start by filling in the parameters. Then, you'll authenticate to all the data sources this app connects to.

Parameters

Make sure all required (*) parameters are filled in before connecting to your data.

StorageAccount *

The Azure Storage account name you use for Azure Cognitive Search Logs and Metrics. More information on how to enable logging can be found here:
<https://docs.microsoft.com/azure/search/search-monitor-usage>

Days *

Number of days of data to query.

Next

Cancel

7. Select **Key** as the authentication method and provide your storage account key. Select **None** or **Private** as the privacy level. Select **Sign In** to begin the loading process.

You are connecting to

Account

Domain



Authentication method

Key

Account key

< Enter your storage account key here >

Privacy level setting for this data source [Learn more](#)

None

Back

Sign in and connect

Cancel

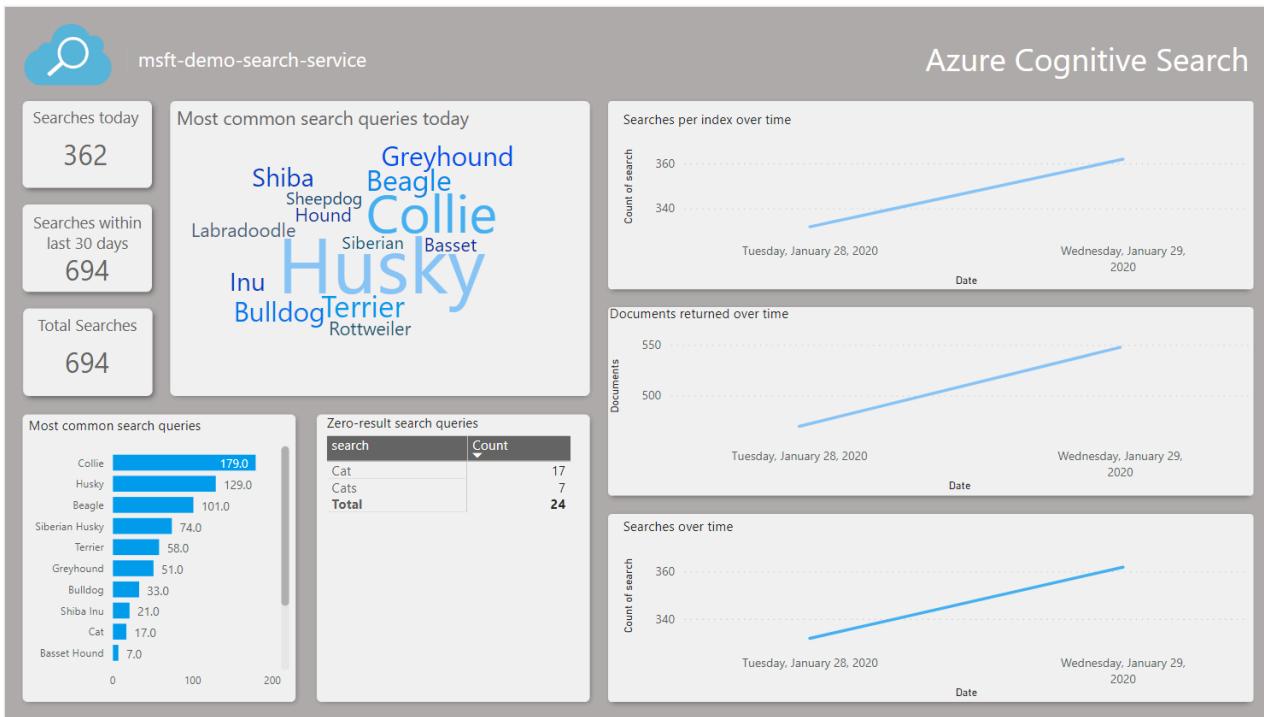
8. Wait for the data to refresh. This might take some time depending on how much data you have. You can see if the data is still being refreshed based on the below indicator.

Name	Type	Sensitivity (preview)	Owner	Refreshed	Endorsement	Include in app
Azure Cognitive Search Dataset	Dataset	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	
Azure Cognitive Search Report	Report	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	Yes

9. Select **Azure Cognitive Search Report** to view the report.

Name	Type	Sensitivity (preview)	Owner	Refreshed	Endorsement	Include in app
Azure Cognitive Search Dataset	Dataset	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	
Azure Cognitive Search Report	Report	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	Yes

10. Refresh the page after opening the report so that it opens with your data.



Modify app parameters

If you would like to visualize data from a different storage account or change the number of days of data to query, follow the below steps to change the **Days** and **StorageAccount** parameters.

1. Navigate to your Power BI apps, find your search app, and select the **Edit** action to continue to the workspace.
2. Select **Settings** from the Dataset options.

Name	Type	Sensitivity (preview)
 Azure Cognitive Search Dataset	Dataset	—
 Azure Cognitive Search Report		<ul style="list-style-type: none"> Analyze in Excel Create report Delete Get quick insights Refresh now Rename Schedule refresh Settings Download .pbix Manage permissions View related

3. While in the Datasets tab, change the parameter values and select **Apply**. If there's an issue with the connection, update the data source credentials on the same page.
4. Navigate back to the workspace and select **Refresh now** from the Dataset options.

Name	Type	Sensitivity (preview)
 Azure Cognitive Search Dataset	Dataset	—
 Azure Cognitive Search Report		<ul style="list-style-type: none"> Analyze in Excel Create report Delete Get quick insights Refresh now Rename Schedule refresh Settings Download .pbix Manage permissions View related

5. Open the report to view the updated data. You might also need to refresh the report to view the latest data.

Troubleshooting report issues

If you can't see your data, try these troubleshooting steps:

1. Open the report and refresh the page to make sure you're viewing the latest data. There's an option in the report to refresh the data. Select this to get the latest data.
2. Ensure the storage account name and access key you provided are correct. The storage account name should correspond to the account configured with your search service logs.
3. Confirm that your storage account contains the containers **insights-logs-operationlogs** and **insights-metrics-pt1m** and each container has data. The logs and metrics will be within a couple layers of folders.
4. Check to see if the dataset is still refreshing. The refresh status indicator is shown in step 8 above. If it's still refreshing, wait until the refresh is complete to open and refresh the report.

Next steps

- [Monitor search operations and activity](#)
- [What is Power BI?](#)
- [Basic concepts for designers in the Power BI service](#)

Analyze performance in Azure AI Search

Article • 01/16/2025

This article describes the tools, behaviors, and approaches for analyzing query and indexing performance in Azure AI Search.

Develop baseline numbers

In any large implementation, it's critical to do a performance benchmarking test of your Azure AI Search service before you roll it into production. You should test both the search query load that you expect, but also the expected data ingestion workloads (if possible, run both workloads simultaneously). Having benchmark numbers helps to validate the proper [search tier](#), [service configuration](#), and expected [query latency](#).

To isolate the effects of a distributed service architecture, try testing on service configurations of one replica and one partition.

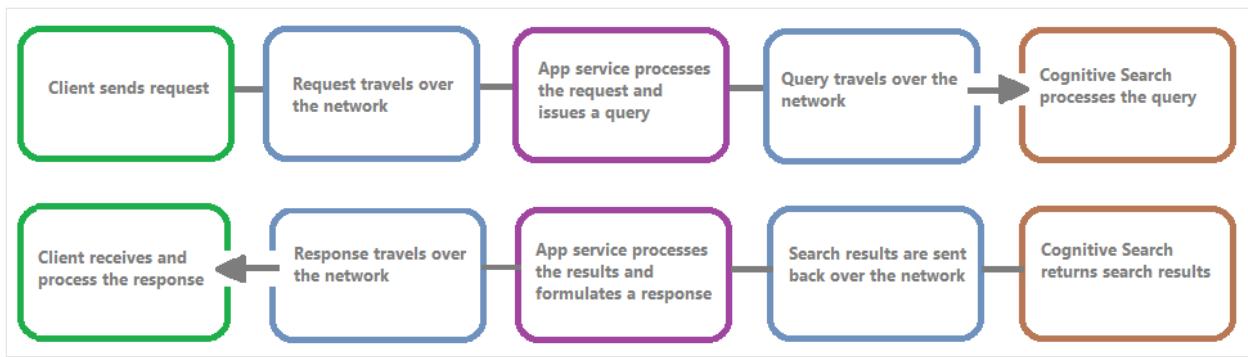
 **Note**

For the Storage Optimized tiers (L1 and L2), you should expect a lower query throughput and higher latency than the Standard tiers.

Use resource logging

The most important diagnostic tool at an administrator's disposal is [resource logging](#). Resource logging is the collection of operational data and metrics about your search service. Resource logging is enabled through [Azure Monitor](#). There are costs associated with using Azure Monitor and storing data, but if you enable it for your service, it can be instrumental in investigating performance issues.

The following image shows the chain of events in a query request and response. Latency can occur at any one of them, whether during a network transfer, processing of content in the app services layer, or on a search service. A key benefit of resource logging is that activities are logged from the search service perspective, which means that the log can help you determine if the performance issue is due to problems with the query or indexing, or some other point of failure.



Resource logging gives you options for storing logged information. We recommend using [Log Analytics](#) so that you can execute advanced Kusto queries against the data to answer many questions about usage and performance.

On your search service portal pages, you can enable logging through [Diagnostic settings](#), and then issue Kusto queries against Log Analytics by choosing [Logs](#). To learn how to send resource logs to a Log Analytics workspace where you can analyze them with log queries, see [Collect and analyze resource logs from an Azure resource](#).

Throttling behaviors

Throttling occurs when the search service is at capacity. Throttling can occur during queries or indexing. From the client side, an API call results in a 503 HTTP response when it has been throttled. During indexing, there's also the possibility of receiving a 207 HTTP response, which indicates that one or more items failed to index. This error is an indicator that the search service is getting close to capacity.

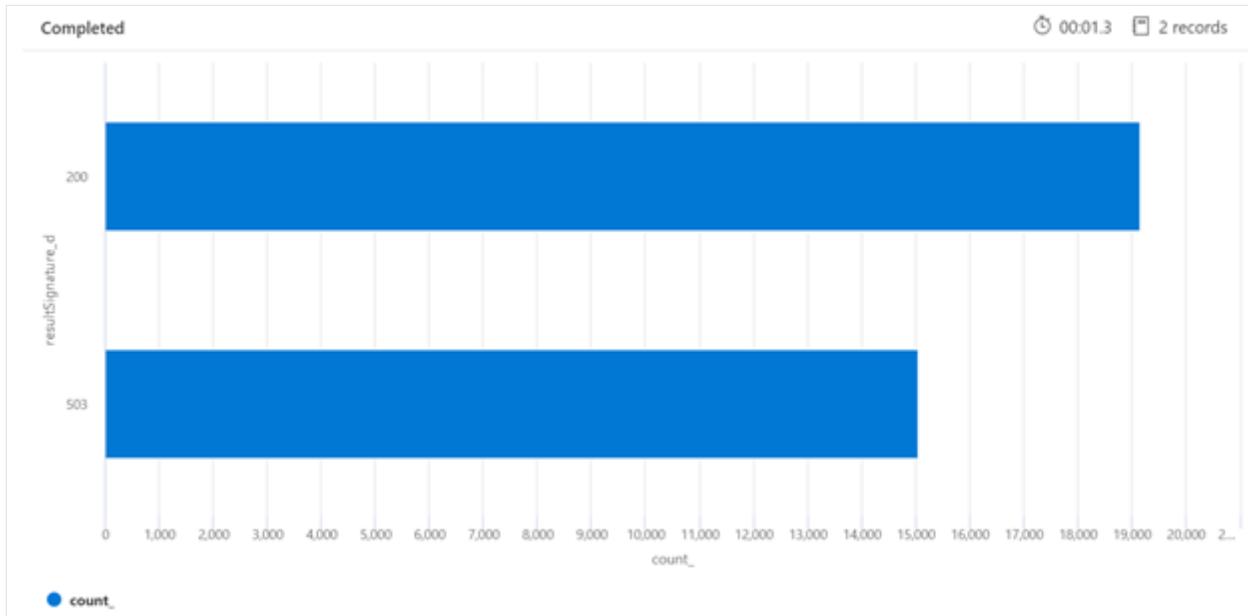
As a rule of thumb, try to quantify the amount of throttling and any patterns. For example, if one search query out of 500,000 is throttled, it might not be worth investigating. However, if a large percentage of queries is throttled over a period, this would be a greater concern. By looking at throttling over a period, it also helps to identify time frames where throttling might more likely occur and help you decide how to best accommodate that.

A simple fix to most throttling issues is to throw more resources at the search service (typically replicas for query-based throttling, or partitions for indexing-based throttling). However, increasing replicas or partitions adds cost, which is why it's important to know the reason why throttling is occurring at all. Investigating the conditions that cause throttling will be explained in the next several sections.

Below is an example of a Kusto query that can identify the breakdown of HTTP responses from the search service that has been under load. Over a 7-day period, the rendered bar chart shows that a relatively large percentage of the search queries were throttled, in comparison to the number of successful (200) responses.

Kusto

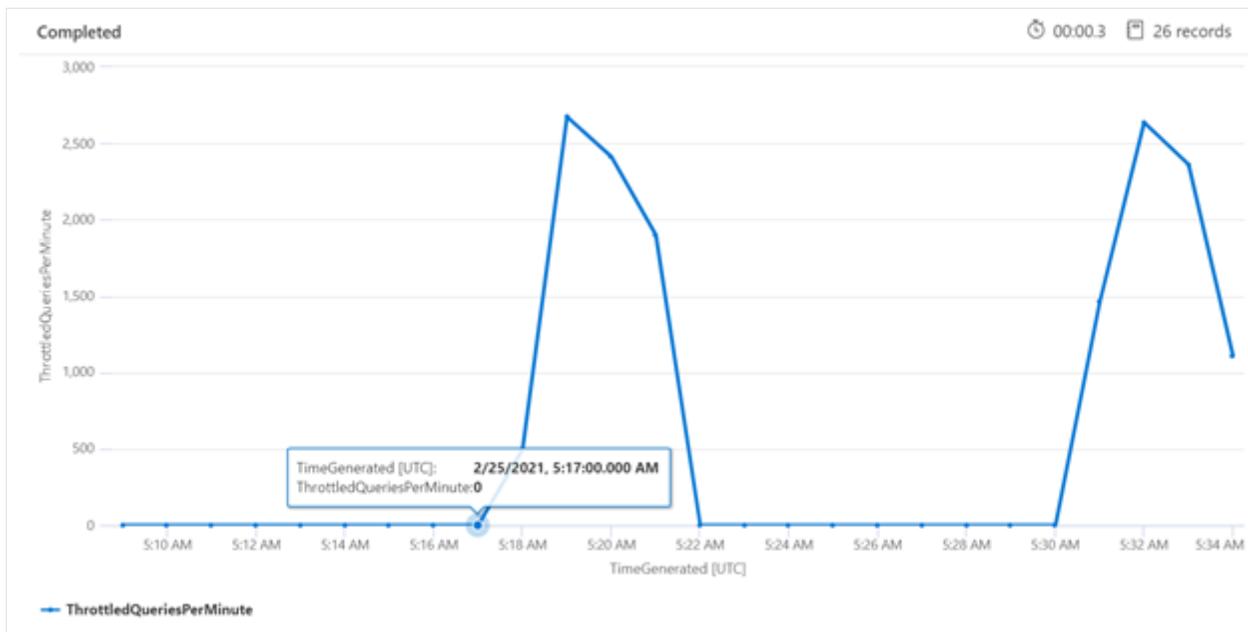
```
AzureDiagnostics
| where TimeGenerated > ago(7d)
| summarize count() by resultSignature_d
| render barchart
```



Examining throttling over a specific time period can help you identify the times where throttling might occur more frequently. In the below example, a time series chart is used to show the number of throttled queries that occurred over a specified time frame. In this case, the throttled queries correlated with the times in with the performance benchmarking was performed.

Kusto

```
let [_startTime]=datetime('2024-02-25T20:45:07Z');
let [_endTime]=datetime('2024-03-03T20:45:07Z');
let intervalsize = 1m;
AzureDiagnostics
| where TimeGenerated > ago(7d)
| where resultSignature_d != 403 and resultSignature_d != 404 and
OperationName in ("Query.Search", "Query.Suggest", "Query.Lookup",
"Query.Autocomplete")
| summarize
    ThrottledQueriesPerMinute=bin(countif(OperationName in ("Query.Search",
"Query.Suggest", "Query.Lookup", "Query.Autocomplete") and resultSignature_d
== 503)/(intervalsize/1m), 0.01)
    by bin(TimeGenerated, intervalsize)
| render timechart
```



Measure individual queries

In some cases, it can be useful to test individual queries to see how they're performing. To do this, it's important to be able to see how long the search service takes to complete the work, as well as how long it takes to make the round-trip request from the client and back to the client. The diagnostics logs could be used to look up individual operations, but it might be easier to do this all from a REST client.

In the example below, a REST-based search query was executed. Azure AI Search includes in every response the number of milliseconds it takes to complete the query, visible in the Headers tab, in "elapsed-time". Next to Status at the top of the response, you'll find the round-trip duration, in this case, 418 milliseconds (ms). In the results section, the "Headers" tab was chosen. Using these two values, highlighted with a red box in the image below, we see the search service took 21 ms to complete the search query and the entire client round-trip request took 125 ms. By subtracting these two numbers we can determine that it took 104-ms additional time to transmit the search query to the search service and to transfer the search results back to the client.

This technique helps you isolate network latencies from other factors impacting query performance.

The screenshot shows a POST request to `https://.search.windows.net/indexes/test/docs?api-version=2020-06-30&search="quick brown foxes"`. The request includes two query parameters: `api-version` (value: 2020-06-30) and `search` (value: "quick brown foxes"). The response status is 200 OK, time: 125 ms, and size: 709 B. A red box highlights the `elapsed-time` header value of 21.

Header	Value
Cache-Control	no-cache
Pragma	no-cache
Content-Type	application/json; odata.metadata=minimal
Content-Encoding	gzip
Expires	-1
Vary	Accept-Encoding
request-id	4ba505e3-5d77-4702-8e4e-473a5856cb27
elapsed-time	21
OData-Version	4.0
Preference-Applied	odata.include-annotations="*"
Strict-Transport-Security	max-age=15724800; includeSubDomains
Date	Thu, 04 Mar 2021 00:43:30 GMT
Content-Length	270

Query rates

One potential reason for your search service to throttle requests is due to the sheer number of queries being performed where volume is captured as queries per second (QPS) or queries per minute (QPM). As your search service receives more QPS, it will typically take longer and longer to respond to those queries until it can no longer keep up, at which it will send back a throttling 503 HTTP response.

The following Kusto query shows query volume as measured in QPM, along with average duration of a query in milliseconds (AvgDurationMS) and the average number of documents (AvgDocCountReturned) returned in each one.

```

Kusto

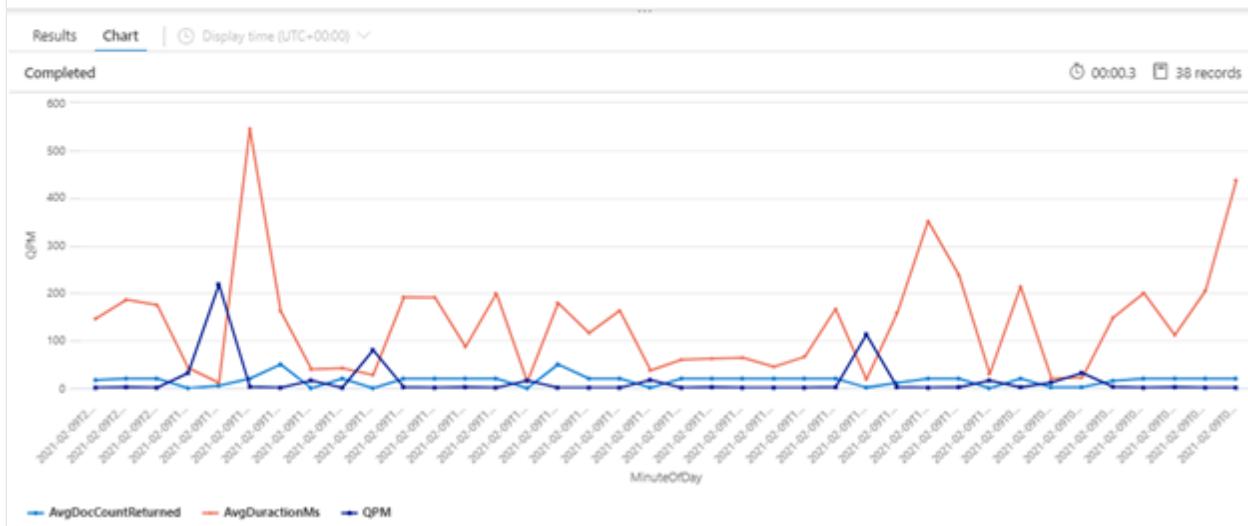
AzureDiagnostics
| where OperationName == "Query.Search" and TimeGenerated > ago(1d)
| extend MinuteOfDay = substring(TimeGenerated, 0, 16)
| project MinuteOfDay, DurationMs, Documents_d, IndexName_s
| summarize QPM=count(), AvgDurationMs=avg(DurationMs),
AvgDocCountReturned=avg(Documents_d) by MinuteOfDay
| order by MinuteOfDay desc
| render timechart

```

```

1 AzureDiagnostics
2 | where OperationName == "Query.Search" and TimeGenerated > ago(1d)
3 | extend MinuteOfDay = substring(TimeGenerated, 0, 16)
4 | project MinuteOfDay, DurationMs, Documents_d, IndexName_s
5 | summarize QPM=count(), AvgDurationMs=avg(DurationMs), AvgDocCountReturned=avg(Documents_d) by MinuteOfDay
6 | order by MinuteOfDay desc
7 | render timechart
8

```



💡 Tip

To reveal the data behind this chart, remove the line `| render timechart` and then rerun the query.

Impact of indexing on queries

An important factor to consider when looking at performance is that indexing uses the same resources as search queries. If you're indexing a large amount of content, you can expect to see latency grow as the service tries to accommodate both workloads.

If queries are slowing down, look at the timing of indexing activity to see if it coincides with query degradation. For example, perhaps an indexer is running a daily or hourly job that correlates with the decreased performance of the search queries.

This section provides a set of queries that can help you visualize the search and indexing rates. For these examples, the time range is set in the query. Be sure to indicate **Set in query** when running the queries in Azure portal.

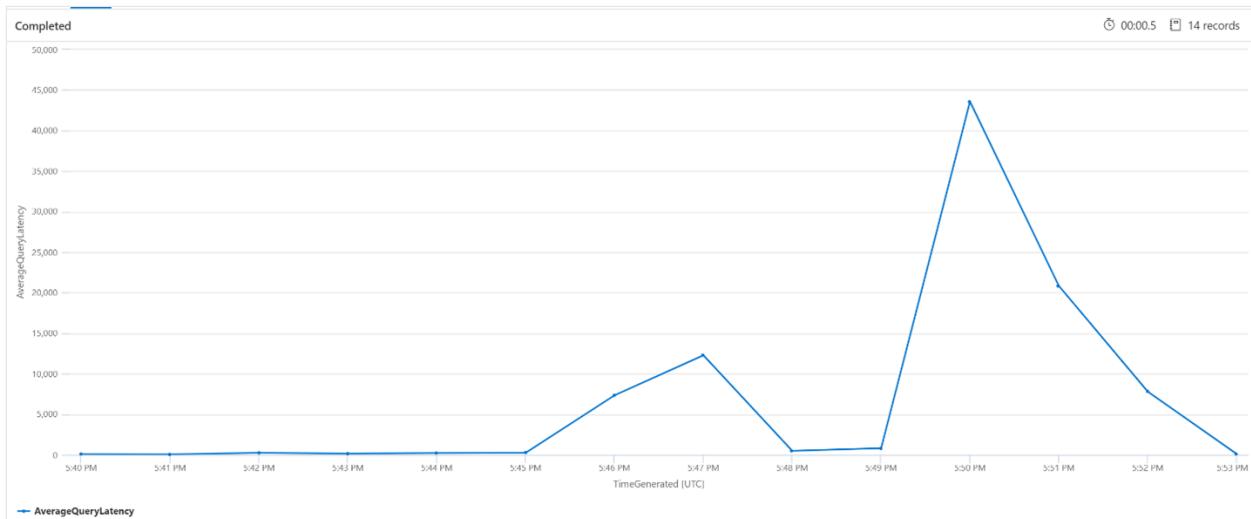


Average Query Latency

In the below query, an interval size of 1 minute is used to show the average latency of the search queries. From the chart, we can see that the average latency was low until 5:45pm and lasted until 5:53pm.

Kusto

```
let intervalsize = 1m;
let _startTime = datetime('2024-02-23 17:40');
let _endTime = datetime('2024-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between(['_startTime']..['_endTime']) // Time range filtering
| summarize AverageQueryLatency = avg(DurationMs, OperationName in ("Query.Search", "Query.Suggest", "Query.Lookup", "Query.AutoComplete"))
    by bin(TimeGenerated, intervalsize)
| render timechart
```

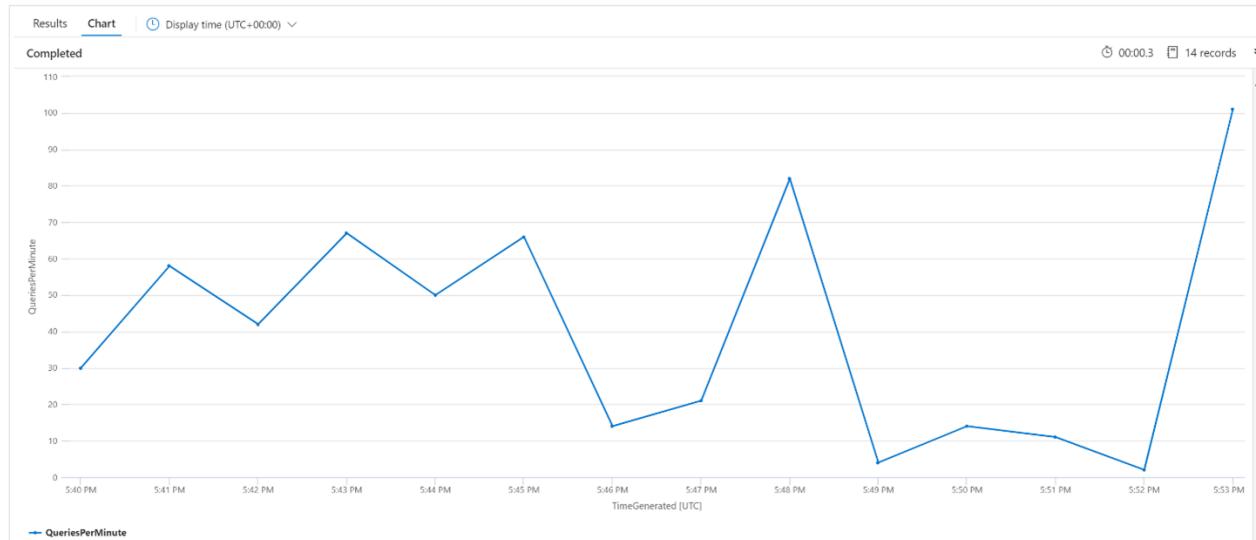


Average Queries Per Minute (QPM)

The following query looks at the average number of queries per minute to ensure that there wasn't a spike in search requests that might have affected the latency. From the chart, we can see there's some variance, but nothing to indicate a spike in request count.

```
Kusto

let intervalsize = 1m;
let _startTime = datetime('2024-02-23 17:40');
let _endTime = datetime('2024-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between([ '_startTime'] .. [ '_endTime']) // Time range filtering
| summarize QueriesPerMinute=bin(countif(OperationName in ("Query.Search", "Query.Suggest", "Query.Lookup", "Query.AutoComplete"))/(intervalsize/1m), 0.01)
    by bin(TimeGenerated, intervalsize)
| render timechart
```



Indexing Operations Per Minute (OPM)

Here we'll look at the number of Indexing operations per minute. From the chart, we can see that a large amount of data was indexed started at 5:42 pm and ended at 5:50pm. This indexing began 3 minutes before the search queries started becoming latent and ended 3 minutes before the search queries were no longer latent.

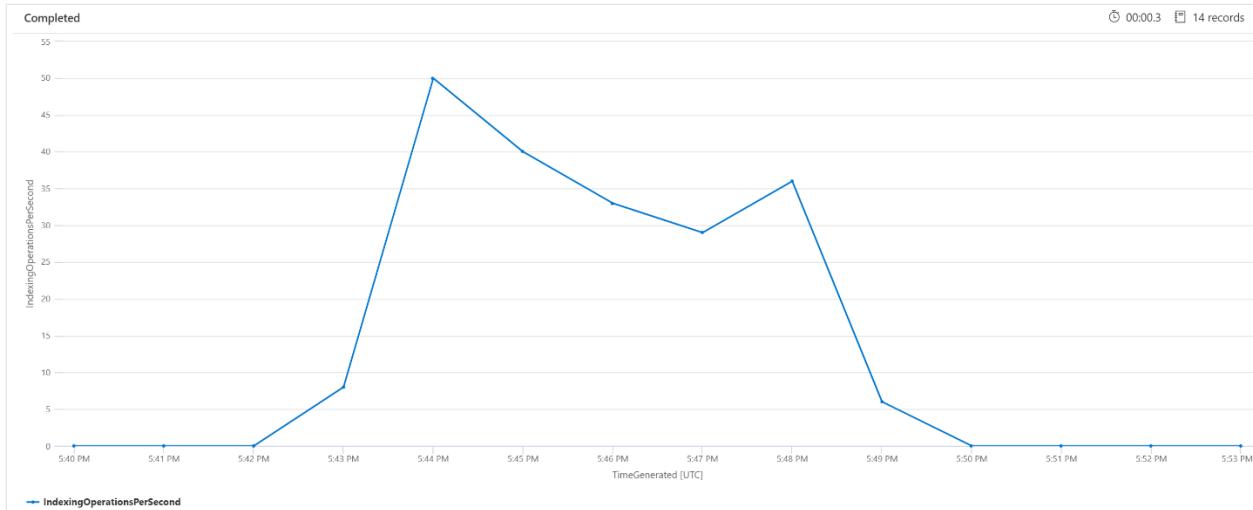
From this insight, we can see that it took about 3 minutes for the search service to become busy enough for indexing to affect query latency. We can also see that after indexing completed, it took another 3 minutes for the search service to complete all the work from the newly indexed content, and for query latency to resolve.

```
Kusto
```

```

let intervalsize = 1m;
let _startTime = datetime('2024-02-23 17:40');
let _endTime = datetime('2024-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between([ '_startTime' ] .. [ '_endTime' ]) // Time range
filtering
| summarize IndexingOperationsPerSecond=bin(countif(OperationName ==
"Indexing.Index")/ (intervalsize/1m), 0.01)
    by bin(TimeGenerated, intervalsize)
| render timechart

```



Background service processing

It's common to see occasional spikes in query or indexing latency. Spikes might occur in response to indexing or high query rates, but could also occur during merge operations. Search indexes are stored in chunks - or shards. Periodically, the system merges smaller shards into large shards, which can help optimize service performance. This merge process also cleans up documents that have previously been marked for deletion from the index, resulting in the recovery of storage space.

Merging shards is fast, but also resource intensive and thus has the potential to degrade service performance. If you notice short bursts of query latency, and those bursts coincide with recent changes to indexed content, you can assume the latency is due to shard merge operations.

Next steps

Review these articles related to analyzing service performance.

- [Performance tips](#)
- [Choose a service tier](#)

- Manage capacity
 - Case Study: Use Cognitive Search to Support Complex AI Scenarios ↗
-

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback ↗ | Get help at Microsoft Q&A

Tips for better performance in Azure AI Search

08/01/2025

This article is a collection of tips and best practices for boosting query and indexing performance for keyword search. Knowing which factors are most likely to affect search performance can help you avoid inefficiencies and get the most out of your search service. Some key factors include:

- Index composition (schema and size)
- Query design
- Service capacity (tier, and the number of replicas and partitions)

! Note

Looking for strategies on high volume indexing? See [Index large data sets in Azure AI Search](#).

Index size and schema

Queries run faster on smaller indexes. This is partly a function of having fewer fields to scan, but it's also due to how the system caches content for future queries. After the first query, some content remains in memory where it's searched more efficiently. Because index size tends to grow over time, one best practice is to periodically revisit index composition, both schema and documents, to look for content reduction opportunities. However, if the index is right-sized, the only other calibration you can make is to increase capacity by [upgrading your service, adding replicas, or switching to a higher pricing tier](#). The section "[Tip: Switch to a Standard S2 tier](#)" discusses the scale up versus scale out decision.

Schema complexity can also adversely affect indexing and query performance. Excessive field attribution builds in limitations and processing requirements. [Complex types](#) take longer to index and query. The next few sections explore each condition.

Tip: Be selective in field attribution

A common mistake that administrators and developers make when creating a search index is selecting all available properties for the fields, as opposed to only selecting just the properties that are needed. For example, if a field doesn't need to be full text searchable, skip that field when setting the searchable attribute.



Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable
HotelId	Edm.String	<input checked="" type="checkbox"/>				
HostName	Edm.String	<input checked="" type="checkbox"/>				
Description	Edm.String	<input checked="" type="checkbox"/>				
Description_fr	Edm.String	<input checked="" type="checkbox"/>				
Category	Edm.String	<input checked="" type="checkbox"/>				
Tags	Collection(Edm.String)	<input checked="" type="checkbox"/>				
ParkingIncluded	Edm.Boolean	<input checked="" type="checkbox"/>				
LastRenovationDate	Edm.DateTime	<input checked="" type="checkbox"/>				
Rating	Edm.Double	<input checked="" type="checkbox"/>				
► Address	Edm.Complex					
Location	Edm.Geogra...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
► Rooms	Collection(Edm.String)					
id	Edm.String	<input checked="" type="checkbox"/>				
rid	Edm.String	<input checked="" type="checkbox"/>				

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable
HotelId	Edm.String	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
HostName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Category	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tags	Collection(Edm.String)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ParkingIncluded	Edm.Boolean	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
LastRenovationDate	Edm.DateTime	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
► Address	Edm.Complex					
Location	Edm.Geogra...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
► Rooms	Collection(Edm.String)					
id	Edm.String	<input type="checkbox"/>				
rid	Edm.String	<input type="checkbox"/>				

Support for filters, facets, and sorting can quadruple storage requirements. If you add suggesters, storage requirements go up even more. For an illustration on the impact of attributes on storage, see [Attributes and index size](#).

Summarized, the ramifications of over-attribution include:

- Degradation of indexing performance due to the extra work required to process the content in the field, and then store it within the search inverted index (set the "searchable" attribute only on fields that contain searchable content).
- Creates a larger surface that each query has to cover. All fields marked as searchable are scanned in a full text search.
- Increases operational costs due to extra storage. Filtering and sorting requires additional space for storing original (non-analyzed) strings. Avoid setting filterable or sortable on fields that don't need it.
- In many cases, over attribution limits the capabilities of the field. For example, if a field is facetable, filterable, and searchable, you can only store 16 KB of text within a field, whereas a searchable field can hold up to 16 MB of text.

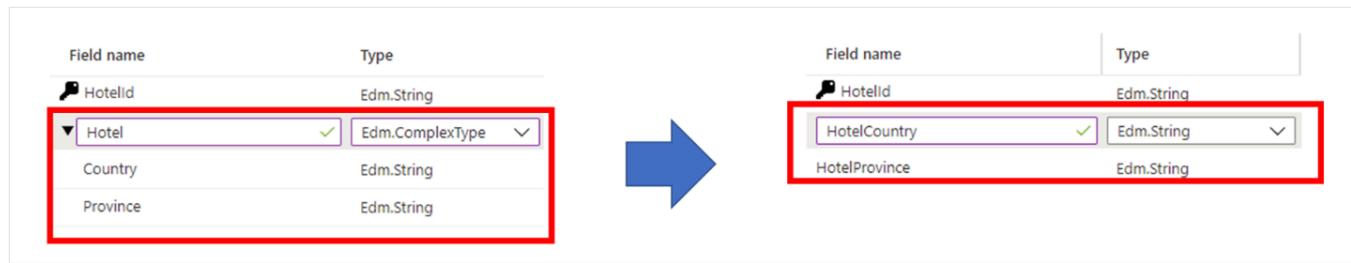
! Note

Only unnecessary attribution should be avoided. Filters and facets are often essential to the search experience, and in cases where filters are used, you frequently need sorting so that you can order the results (filters by themselves return in an unordered set).

Tip: Consider alternatives to complex types

Complex data types are useful when data has a complicated nested structure, such as the parent-child elements found in JSON documents. The downside of complex types is the extra storage requirements and additional resources required to index the content, in comparison to non-complex data types.

In some cases, you can avoid these tradeoffs by mapping a complex data structure to a simpler field type, such as a Collection. Alternatively, you might opt for flattening a field hierarchy into individual root-level fields.



Query design

Query composition and complexity are one of the most important factors for performance, and query optimization can drastically improve performance. When designing queries, think about the following points:

- **Number of searchable fields.** Each additional searchable field results in more work for the search service. You can limit the fields being searched at query time using the "searchFields" parameter. It's best to specify only the fields that you care about to improve performance.
- **Amount of data being returned.** Retrieving a large amount content can make queries slower. When structuring a query, return only those fields that you need to render the results page, and then retrieve remaining fields using the [Lookup API](#) once a user selects a match.
- **Use of partial term searches.** [Partial term searches](#), such as prefix search, fuzzy search, and regular expression search, are more computationally expensive than typical keyword searches, as they require full index scans to produce results.
- **Number of facets.** Adding facets to queries requires aggregations for each query. Requesting a higher "count" for a facet also requires extra work by the service. In general, only add the facets that you plan to render in your app and avoid requesting a high count for facets unless necessary.
- **High skip values.** Setting the `$skip` parameter to a high value (for example, in the thousands) increases search latency because the engine is retrieving and ranking a larger volume of documents for each request. For performance reasons, it's best to avoid high `$skip` values and use other techniques instead, such as filtering, to retrieve large numbers of documents.

- **Limit high cardinality fields.** A *high cardinality field* refers to a facetable or filterable field that has a significant number of unique values, and as a result, consumes significant resources when computing results. For example, setting a Product ID or Description field as facetable and filterable would count as high cardinality because most of the values from document to document are unique.

Tip: Use search functions instead overloading filter criteria

As a query uses increasingly [complex filter criteria](#), the performance of the search query will degrade. Consider the following example that demonstrates the use of filters to trim results based on a user identity:

JSON

```
$filter= userid eq 123 or userid eq 234 or userid eq 345 or userid eq 456 or  
userid eq 567
```

In this case, the filter expressions are used to check whether a single field in each document is equal to one of many possible values of a user identity. You're most likely to find this pattern in applications that implement [security trimming](#) (checking a field containing one or more principal IDs against a list of principal IDs representing the user issuing the query).

A more efficient way to execute filters that contain a large number of values is to use [search.in function](#), as shown in this example:

JSON

```
search.in(userid, '123,234,345,456,567', ',')
```

Tip: Add partitions for slow individual queries

When query performance is slowing down in general, adding more replicas frequently solves the issue. But what if the problem is a single query that takes too long to complete? In this scenario, adding replicas won't help, but more partitions might. A partition splits data across extra computing resources. Two partitions split data in half, a third partition splits it into thirds, and so forth.

One positive side-effect of adding partitions is that slower queries sometimes perform faster due to parallel computing. We've noted parallelization on low selectivity queries, such as queries that match many documents, or facets providing counts over a large number of documents. Since significant computation is required to score the relevancy of the documents, or to count the numbers of documents, adding extra partitions helps queries complete faster.

To add partitions, use the [Azure portal](#), [PowerShell](#), [Azure CLI](#), or a management SDK.

Service capacity

A service is overburdened when queries take too long or when the service starts dropping requests. If this happens, you can address the problem by upgrading the service or by adding capacity.

The tier of your search service and the number of replicas/partitions also have a large impact on performance. Each progressively higher tier provides faster CPUs and more memory, both of which have a positive impact on performance.

Tip: Create a new high capacity search service

Basic and Standard services created [in supported regions](#) after April 3, 2024 have more storage per partition than older services. If you have an older service, check if you can [upgrade your service](#) to benefit from more capacity at the same billing rate. If an upgrade isn't available, review the [tier service limits](#) to see if the same tier on a newer service gives you the necessary storage.

Tip: Switch to a Standard S2 tier

The Standard S1 search tier is often where customers start. A common pattern for S1 services is that indexes grow over time, which requires more partitions. More partitions lead to slower response times, so more replicas are added to handle the query load. As you can imagine, the cost of running an S1 service has now progressed to levels beyond the initial configuration.

At this juncture, an important question to ask is whether it would be beneficial to [switch to a higher pricing tier](#), as opposed to progressively increasing the number of partitions or replicas of the current service.

Consider the following topology as an example of a service that has taken on increasing levels of capacity:

- Standard S1 tier
- Index Size: 190 GB
- Partition Count: 8 (on S1, partition size is 25 GB per partition)
- Replica Count: 2
- Total Search Units: 16 (8 partitions x 2 replicas)
- Hypothetical Retail Price: ~\$4,000 USD / month (assume 250 USD x 16 search units)

Suppose the service administrator is still seeing higher latency rates and is considering adding another replica. This would change the replica count from 2 to 3 and as a result change the Search Unit count to 24 and a resulting price of \$6,000 USD/month.

However, if the administrator chose to move to a Standard S2 tier the topology would look like:

- Standard S2 tier
- Index Size: 190 GB
- Partition Count: 2 (on S2, partition size is 100 GB per partition)
- Replica Count: 2
- Total Search Units: 4 (2 partitions x 2 replicas)
- Hypothetical Retail Price: ~\$4,000 USD / month (1,000 USD x 4 search units)

As this hypothetical scenario illustrates, you can have configurations on lower tiers that result in similar costs as if you had opted for a higher tier in the first place. However, higher tiers come with premium storage, which makes indexing faster. Higher tiers also have much more compute power, as well as extra memory. For the same costs, you could have more powerful infrastructure backing the same index.

An important benefit of added memory is that more of the index can be cached, resulting in lower search latency, and a greater number of queries per second. With this extra power, the administrator might not need to even need to increase the replica count and could potentially pay less than by staying on the S1 service.

Tip: Consider alternatives to regular expression queries

[Regular expression queries](#) or regex can be particularly expensive. While they can be very useful for advanced searches, execution can require a lot of processing power, especially if the regular expression is complicated or if you're searching through a large amount of data. All of these factors contribute to high search latency. As a mitigation, try to simplify the regular expression or break the complex query down into smaller, more manageable queries.

Next steps

Review these other articles related to service performance:

- [Analyze performance](#)
- [Index large data sets in Azure AI Search](#)
- [Choose a service tier](#)
- [Plan or add capacity](#)
- [Case Study: Use Cognitive Search to Support Complex AI Scenarios ↗](#)

Knowledge store "projections" in Azure AI Search

! Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in agentic retrieval workflows.

Projections define the physical tables, objects, and files in a **knowledge store** that accept content from an Azure AI Search enrichment pipeline. If you're creating a knowledge store, defining and shaping projections is most of the work.

This article introduces projection concepts and workflow so that you have some background before you start coding.

Projections are defined in Azure AI Search skillsets, but the end results are the table, object, and image file projections in Azure Storage.

The screenshot shows the Azure Storage browser (preview) interface for a storage account named 'demoblobstorage'. The left sidebar has a 'Storage browser (preview)' section highlighted with a red box. The main area shows a list of storage entities: blobstorage, Favorites, Recently viewed, Blob containers, File shares, Queues, and Tables. The 'Tables' item is also highlighted with a red box. On the right, a table list is displayed with three entries highlighted by a red box: 'kstoreProjectionDemoDocument', 'kstoreProjectionDemoEntities', and 'kstoreProjectionDemoKeyPhrases'. A search bar and a message about authentication method are also visible.

Types of projections and usage

A knowledge store is a logical construction that's physically expressed as a loose collection of tables, JSON objects, or binary image files in Azure Storage.

Projection	Storage	Usage
Tables	Azure Table Storage	Used for data that's best represented as rows and columns, or whenever you need granular representations of your data (for example, as data frames). Table projections allow you to define a schematized shape, using a Shaper skill or use inline shaping to specify columns and rows. You can organize content into multiple tables based on familiar normalization principles. Tables that are in the same group are automatically related.
Objects	Azure Blob Storage	Used when you need the full JSON representation of your data and enrichments in one JSON document. As with table projections, only valid JSON objects can be projected as objects, and shaping can help you do that.
Files	Azure Blob Storage	Used when you need to save normalized, binary image files.

Projection definition

Projections are specified under the "knowledgeStore" property of a [skillset](#). Projection definitions are used during indexer invocation to create and load objects in Azure Storage with enriched content. If you're unfamiliar with these concepts, start with [AI enrichment](#) for an introduction.

The following example illustrates the placement of projections under knowledgeStore, and the basic construction. The name, type, and content source make up a projection definition.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                { "tableName": "ks-museums-main", "generatedKeyName": "ID", "source": "/document/tableprojection" },
                { "tableName": "ks-museumEntities", "generatedKeyName": "ID", "source": "/document/tableprojection/Entities/*" }
            ],
            "objects": [
                { "storageContainer": "ks-museums", "generatedKeyName": "ID", "source": "/document/objectprojection" }
            ],
            "files": [ ]
```

```
}
```

Projection groups

Projections are an array of complex collections, which means that you can specify multiple sets of each type. It's common to use just one projection group, but you might use multiple if storage requirements include supporting different tools and scenarios. For example, you might use one group for design and debug of a skillset, while a second set collects output used for an online app, with a third for data science workloads.

The same skillset output is used to populate all groups under projections. The following example shows two.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct
Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [],
            "objects": [],
            "files": []
        },
        {
            "tables": [],
            "objects": [],
            "files": []
        }
    ]
}
```

Projection groups have the following key characteristics of mutual exclusivity and relatedness.

 Expand table

Principle	Description
Mutual exclusivity	Each group is fully isolated from other groups to support different data shaping scenarios. For example, if you're testing different table structures and combinations, you would put each set in a different projection group for AB testing. Each group obtains data from the same source (enrichment tree) but is fully isolated from the table-object-file combination of any peer projection groups.
Relatedness	Within a projection group, content in tables, objects, and files are related. Knowledge store uses generated keys as reference points to a common parent node. For example, consider a scenario where you have a document containing images and text. You could

Principle	Description
	project the text to tables and the images to binary files, and both tables and objects have a column/property containing the file URL.

Projection "source"

The source parameter is the third component of a projection definition. Because projections store data from an AI enrichment pipeline, the source of a projection is always the output of a skill. As such, output might be a single field (for example, a field of translated text), but often it's a reference to a data shape.

Data shapes come from your skillset. Among all of the built-in skills provided in Azure AI Search, there's a utility skill called the [Shaper skill](#) that's used to create data shapes. You can include Shaper skills (as many as you need) to support the projections in the knowledge store.

Shapes are frequently used with table projections, where the shape not only specifies which rows go into the table, but also which columns are created (you can also pass a shape to an object projection).

Shapes can be complex and it's out of scope to discuss them in depth here, but the following example briefly illustrates a basic shape. The output of the Shaper skill is specified as the source of a table projection. Within the table projection itself will be columns for "metadata_storage_path", "reviews_text", "reviews_title", and so forth, as specified in the shape.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "ShaperForTables",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "metadata_storage_path",
      "source": "/document/metadata_storage_path",
      "sourceContext": null,
      "inputs": []
    },
    {
      "name": "reviews_text",
      "source": "/document/reviews_text"
    },
    {
      "name": "reviews_title",
      "source": "/document/reviews_title"
    }
  ]
}
```

```
        "name": "reviews_username",
        "source": "/document/reviews_username"
    },
],
"outputs": [
{
    "name": "output",
    "targetName": "mytableprojection"
}
]
}
```

Projection lifecycle

Projections have a lifecycle that is tied to the source data in your data source. As source data is updated and reindexed, projections are updated with the results of the enrichments, ensuring your projections are eventually consistent with the data in your data source. However, projections are also independently stored in Azure Storage. They won't be deleted when the indexer or the search service itself is deleted.

Consume in apps

After the indexer is run, connect to projections and consume the data in other apps and workloads.

- Use Azure portal to verify object creation and content in Azure Storage.
- Use [Power BI for data exploration](#). This tool works best when the data is in Azure Table Storage. Within Power BI, you can manipulate data into new tables that are easier to query and analyze.
- Use enriched data in blob container in a data science pipeline. For example, you can [load the data from blobs into a Pandas DataFrame](#).
- Finally, if you need to export your data from the knowledge store, Azure Data Factory has connectors to export the data and land it in the database of your choice.

Checklist for getting started

Recall that projections are exclusive to knowledge stores, and aren't used to structure a search index.

1. In Azure Storage, get a connection string from [Access Keys](#) and verify the account is StorageV2 (general purpose V2).

2. While in Azure Storage, familiarize yourself with existing content in containers and tables so that you choose nonconflicting names for the projections. A knowledge store is a loose collection of tables and containers. Consider adopting a naming convention to keep track of related objects.
3. In Azure AI Search, [enable enrichment caching \(preview\)](#) in the indexer and then [run the indexer](#) to execute the skillset and populate the cache. This is a preview feature, so be sure to use the preview REST API on the indexer request. Once the cache is populated, you can modify projection definitions in a knowledge store free of charge (as long as the skills themselves aren't modified).
4. In your code, all projections are defined solely in a skillset. There are no indexer properties (such as field mappings or output field mappings) that apply to projections. Within a skillset definition, you'll focus on two areas: knowledgeStore property and skills array.
 - a. Under knowledgeStore, specify table, object, file projections in the `projections` section. Object type, object name, and quantity (per the number of projections you define) are determined in this section.
 - b. From the skills array, determine which skill outputs should be referenced in the `source` of each projection. All projections have a source. The source can be the output of an upstream skill, but is often the output of a Shaper skill. The composition of your projection is determined through shapes.
5. If you're adding projections to an existing skillset, [update the skillset](#) and [run the indexer](#).
6. Check your results in Azure Storage. On subsequent runs, avoid naming collisions by deleting objects in Azure Storage or changing project names in the skillset.
7. If you're using [Table projections](#) check [Understanding the Table Service data model](#) and [Scalability and performance targets for Table storage](#) to make sure your data requirements are within Table storage documented limits.

Next steps

Review syntax and examples for each projection type.

[Define projections in a knowledge store](#)

Create a knowledge store using REST

(!) Note

Knowledge stores exist in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge agents, which are used in [agentic retrieval](#) workflows.

In Azure AI Search, a [knowledge store](#) is a repository of [AI-generated content](#) that's used for non-search scenarios. You create the knowledge store using an indexer and skillset, and specify Azure Storage to store the output. After the knowledge store is populated, use tools like [Storage Explorer](#) or [Power BI](#) to explore the content.

In this article, you use the REST API to ingest, enrich, and explore a set of customer reviews of hotel stays in a knowledge store. The knowledge store contains original text content pulled from the source, plus AI-generated content that includes a sentiment score, key phrase extraction, language detection, and text translation of non-English customer comments.

To make the initial data set available, the hotel reviews are first imported into Azure Blob Storage. Post-processing, the results are saved as a knowledge store in Azure Table Storage.

💡 Tip

This article uses REST for detailed explanations of each step. [Download the REST file](#) if you want to just run the commands. Alternatively, you can also [create a knowledge store in Azure portal](#).

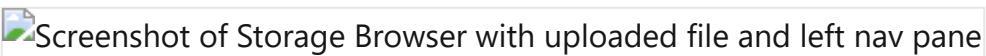
Prerequisites

- Visual Studio Code [with a REST client](#). If you need help with getting started, see [Quickstart: Full-text search using REST](#).
- Azure AI Search. [Create a service](#) or [find an existing one](#). You can use the free service for this exercise.
- Azure Storage. [Create an account](#) or [find an existing one](#). The account type must be **StorageV2 (general purpose V2)**.

The skillset in this example uses Azure AI Services for enrichments. Because the workload is so small, Azure AI services is tapped behind the scenes to provide free processing for up to 20

transactions daily. A small workload means that you can skip creating or attaching an Azure AI services multi-service resource.

Upload data to Azure Storage and get a connection string

1. Download [HotelReviews_Free.csv](#). This CSV contains 19 pieces of customer feedback about a single hotel (originates from Kaggle.com).
 2. In Azure portal, find your storage account and use **Storage Browser** to create a blob container named **hotel-reviews**.
 3. Select **Upload** at the top of the page to load the **HotelReviews-Free.csv** file you downloaded from the previous step.
- Screenshot of Storage Browser with uploaded file and left nav pane
4. On the left, select **Access Keys**, select **Show Keys**, and then copy the connection string for either key1 or key2. A full access connection string has the following format:

JSON

```
"knowledgeStore": {  
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<YOUR-  
ACCOUNT-NAME>;AccountKey=<YOUR-ACCOUNT-KEY>;EndpointSuffix=core.windows.net;"  
}
```

ⓘ Note

See [Connect using a managed identity](#) if you don't want to provide sensitive data on the connection string.

Copy a key and URL

In this example, REST calls require the search service endpoint and use an API key on every request. You can get these values from the Azure portal.

1. Sign in to the [Azure portal](#), navigate to the **Overview** page, and copy the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. Under **Settings > Keys**, copy an admin key. Admin keys are used to add, modify, and delete objects. There are two interchangeable admin keys. Copy either one.

The screenshot displays two windows from the Microsoft Azure portal. The top window is titled 'new-demo-search-svc' and shows the 'Overview' tab selected. It includes a URL field labeled 'Url' with the value 'https://new-demo-search-svc.search.windows.net'. The bottom window is also titled 'new-demo-search-svc' and shows the 'Keys' tab selected. It displays 'API access control' settings with 'API keys' selected. Below this, the 'Manage admin keys' section is shown, featuring fields for 'Primary admin key' and 'Secondary admin key', each containing placeholder text '<your-primary-admin-key-here>' and '<your-secondary-admin-key-here>'. Red boxes and numbers highlight specific areas: 'Overview' at 1, 'Url' at 1, 'Keys' at 2, and the key fields at 2.

A valid API key establishes trust, on a per request basis, between the application sending the request and the search service handling it.

Create an index

[Create Index \(REST\)](#) creates a search index on the search service. A search index is unrelated to a knowledge store, but the indexer requires one. The search index contains the same content as the knowledge store, which you can explore by sending query requests.

1. Open a new text file in Visual Studio Code.
2. Set variables to the search endpoint and the API key you collected earlier.

HTTP

```
@baseUrl = PUT-YOUR-SEARCH-SERVICE-URL-HERE
@apiKey = PUT-YOUR-ADMIN-API-KEY-HERE
@storageConnection = PUT-YOUR-STORAGE-CONNECTION-STRING-HERE
@blobContainer = PUT-YOUR-CONTAINER-NAME-HERE (hotel-reviews)
```

3. Save the file with a `.rest` file extension.
4. Paste in the following example to create the index request.

HTTP

```
### Create a new index
POST {{baseUrl}}/indexes?api-version=2025-09-01 HTTP/1.1
```

```

Content-Type: application/json
api-key: {{apiKey}}


{
    "name": "hotel-reviews-kstore-idx",
    "fields": [
        { "name": "name", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
        { "name": "reviews_date", "type": "Edm.DateTimeOffset",
"searchable": false, "filterable": false, "sortable": false, "facetable": false
},
        { "name": "reviews_rating", "type": "Edm.String", "searchable": false,
"filterable": false, "sortable": false, "facetable": false },
        { "name": "reviews_text", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
        { "name": "reviews_title", "type": "Edm.String", "searchable": false,
"filterable": false, "sortable": false, "facetable": false },
        { "name": "reviews_username", "type": "Edm.String", "searchable": false,
"filterable": false, "sortable": false, "facetable": false },
        { "name": "AzureSearch_DocumentKey", "type": "Edm.String",
"searchable": false, "filterable": false, "sortable": false, "facetable": false,
"key": true },
        { "name": "language", "type": "Edm.String", "filterable": true,
"sortable": false, "facetable": true },
        { "name": "translated_text", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
        { "name": "sentiment", "type": "Collection(Edm.String)",
"searchable": false, "filterable": true, "retrievable": true, "sortable": false,
"facetable": true },
        { "name": "keyphrases", "type": "Collection(Edm.String)",
"filterable": true, "sortable": false, "facetable": true }
    ]
}

```

5. Select **Send request**. You should have an `HTTP/1.1 201 Created` response and the response body should include the JSON representation of the index schema.

Create a data source

[Create Data Source](#) creates a data source connection on Azure AI Search.

- Paste in the following example to create the data source.

HTTP

```

### Create a data source
POST {{baseUrl}}/datasources?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

```

{

```

"name": "hotel-reviews-kstore-ds",
"description": null,
"type": "azureblob",
"subtype": null,
"credentials": {
    "connectionString": "{{storageConnectionString}}"
},
"container": {
    "name": "{{blobContainer}}",
    "query": null
},
"dataChangeDetectionPolicy": null,
"dataDeletionDetectionPolicy": null
}

```

2. Select **Send request**.

Create a skillset

A skillset defines enrichments (skills) and your knowledge store. [Create Skillset](#) creates the object on your search service.

1. Paste in the following example to create the skillset.

HTTP

```

### Create a skillset
POST {{baseUrl}}/skillsets?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}


{
    "name": "hotel-reviews-kstore-ss",
    "description": "Skillset to detect language, translate text, extract key phrases, and score sentiment",
    "skills": [
        {
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
            "context": "/document/reviews_text", "textSplitMode": "pages",
            "maximumPageLength": 5000,
            "inputs": [
                { "name": "text", "source": "/document/reviews_text" }
            ],
            "outputs": [
                { "name": "textItems", "targetName": "pages" }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.V3.SentimentSkill",
            "context": "/document/reviews_text/pages/*",
            "inputs": [

```

```
        { "name": "text", "source":  
"/document/reviews_text/pages/*" },  
            { "name": "languageCode", "source": "/document/language" }  
        ],  
        "outputs": [  
            { "name": "sentiment", "targetName": "sentiment" }  
        ]  
    },  
    {  
        "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",  
        "context": "/document",  
        "inputs": [  
            { "name": "text", "source": "/document/reviews_text" }  
        ],  
        "outputs": [  
            { "name": "languageCode", "targetName": "language" }  
        ]  
    },  
    {  
        "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",  
        "context": "/document/reviews_text/pages/*",  
        "defaultFromLanguageCode": null,  
        "defaultToLanguageCode": "en",  
        "inputs": [  
            { "name": "text", "source":  
"/document/reviews_text/pages/*" }  
        ],  
        "outputs": [  
            { "name": "translatedText", "targetName": "translated_text" }  
        ]  
    },  
    {  
        "@odata.type":  
"#Microsoft.Skills.Text.KeyPhraseExtractionSkill",  
        "context": "/document/reviews_text/pages/*",  
        "inputs": [  
            { "name": "text", "source":  
"/document/reviews_text/pages/*" },  
            { "name": "languageCode", "source": "/document/language" }  
        ],  
        "outputs": [  
            { "name": "keyPhrases" , "targetName": "keyphrases" }  
        ]  
    },  
    {  
        "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",  
        "context": "/document",  
        "inputs": [  
            { "name": "name", "source": "/document/name" },  
            { "name": "reviews_date", "source":  
"/document/reviews_date" },  
            { "name": "reviews_rating", "source":  
"/document/reviews_rating" },  
            { "name": "reviews_text", "source":  
"/document/reviews_text" }  
        ]  
    }  
]
```

```

        "/document/reviews_text" },
            { "name": "reviews_title", "source": "/document/reviews_title" },
            { "name": "reviews_username", "source": "/document/reviews_username" },
            { "name": "AzureSearch_DocumentKey", "source": "/document/AzureSearch_DocumentKey" },
            {
                "name": "pages",
                "sourceContext": "/document/reviews_text/pages/*",
                "inputs": [
                    {
                        "name": "languageCode",
                        "source": "/document/language"
                    },
                    {
                        "name": "translatedText",
                        "source": "/document/reviews_text/pages/*/translated_text"
                    },
                    {
                        "name": "sentiment",
                        "source": "/document/reviews_text/pages/*/sentiment"
                    },
                    {
                        "name": "keyPhrases",
                        "source": "/document/reviews_text/pages/*/keyphrases/*"
                    },
                    {
                        "name": "Page",
                        "source": "/document/reviews_text/pages/*"
                    }
                ]
            }
        ],
        "outputs": [
            { "name": "output" , "targetName": "tableprojection" }
        ]
    }
],
"knowledgeStore": {
    "storageConnectionString": "{{storageConnectionString}}",
    "projections": [
        {
            "tables": [
                { "tableName": "hotelReviews1Document",
"generatedKeyName": "Documentid", "source": "/document/tableprojection" },
                { "tableName": "hotelReviews2Pages",
"generatedKeyName": "Pagesid", "source": "/document/tableprojection/pages/*" },
                { "tableName": "hotelReviews3KeyPhrases",
"generatedKeyName": "KeyPhrasesid", "source": "/document/tableprojection/pages/*/keyPhrases/*" }
            ],
            "objects": []
        },
    ]
}

```

```

{
    "tables": [
        {
            "tableName": "hotelReviews4InlineProjectionDocument", "generatedKeyName": "Documentid",
            "sourceContext": "/document",
            "inputs": [
                { "name": "name", "source": "/document/name" },
                { "name": "reviews_date", "source": "/document/reviews_date" },
                { "name": "reviews_rating", "source": "/document/reviews_rating" },
                { "name": "reviews_username", "source": "/document/reviews_username" },
                { "name": "reviews_title", "source": "/document/reviews_title" },
                { "name": "reviews_text", "source": "/document/reviews_text" },
                { "name": "AzureSearch_DocumentKey", "source": "/document/AzureSearch_DocumentKey" }
            ]
        },
        {
            "tableName": "hotelReviews5InlineProjectionPages",
            "generatedKeyName": "Pagesid", "sourceContext": "/document/reviews_text/pages/*",
            "inputs": [
                { "name": "Sentiment", "source": "/document/reviews_text/pages/*/sentiment" },
                { "name": "LanguageCode", "source": "/document/language" },
                { "name": "Keyphrases", "source": "/document/reviews_text/pages/*/keyphrases" },
                { "name": "TranslatedText", "source": "/document/reviews_text/pages/*/translated_text" },
                { "name": "Page", "source": "/document/reviews_text/pages/*" }
            ]
        },
        {
            "tableName": "hotelReviews6InlineProjectionKeyPhrases", "generatedKeyName": "kpidv2",
            "sourceContext": "/document/reviews_text/pages/*/keyphrases/*",
            "inputs": [
                { "name": "Keyphrases", "source": "/document/reviews_text/pages/*/keyphrases/*" }
            ]
        },
        "objects": []
    ]
}

```

Key points:

- The **Shaper skill** is important to knowledge store definition. It specifies how the data flows into the tables of the knowledge store. The inputs are the parts of the enriched document that you want to store. The output is a consolidation of the nodes into a single structure.
- Projections specify the tables, objects, and blobs of your knowledge store. Each projection item specifies the `"name"` of column or field to create in Azure Storage. The `"source"` specifies which part of the shaper output is assigned to that field or column.

Create an indexer

[Create Indexer](#) creates and runs the indexer. Indexer execution starts by cracking the documents, extracting text and images, and initializing the skillset. The indexer checks for the other objects that you created: the datasource, the index, and the skillset.

1. Paste in the following example to create the indexer.

HTTP

```
### Create indexer
POST {{baseUrl}}/indexers?api-version=2025-09-01 HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

{
    "name": "hotel-reviews-kstore-idxr",
    "dataSourceName": "hotel-reviews-kstore-ds",
    "skillsetName": "hotel-reviews-kstore-ss",
    "targetIndexName": "hotel-reviews-kstore-idx",
    "parameters": {
        "configuration": {
            "dataToExtract": "contentAndMetadata",
            "parsingMode": "delimitedText",
            "firstLineContainsHeaders": true,
            "delimitedTextDelimiter": ","
        }
    },
    "fieldMappings": [
        {
            "sourceFieldName": "AzureSearch_DocumentKey",
            "targetFieldName": "AzureSearch_DocumentKey",
            "mappingFunction": { "name": "base64Encode" }
        }
    ],
    "outputFieldMappings": [
        { "sourceFieldName": "/document/reviews_text/pages/*/Keyphrases/*",
        "targetFieldName": "Keyphrases" },
        { "sourceFieldName": "/document/Language", "targetFieldName": "Language" },
    ]
}
```

```
        { "sourceFieldName": "/document/reviews_text/pages/*/Sentiment",
  "targetFieldName": "Sentiment" }
]
```

2. Select **Send request** to create and run the indexer. This step takes several minutes to complete.

Key points:

- The `parameters/configuration` object controls how the indexer ingests the data. In this case, the input data is in a single CSV file that has a header line and comma-separated values.
- Field mappings create "AzureSearch_DocumentKey" is a unique identifier for each document that's generated by the blob indexer (based on metadata storage path).
- Output field mappings specify how enriched fields are mapped to fields in a search index. Output field mappings aren't used in knowledge stores (knowledge stores use shapes and projections to express the physical data structures).

Check status

After you send each request, the search service should respond with a 201 success message.

HTTP

```
### Get Indexer Status (wait several minutes for the indexer to complete)
GET {{baseUrl}}/indexers/hotel-reviews-kstore-idxr/status?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}
```

After several minutes, you can query the index to inspect the content. Even if you're not using the index, this step is a convenient way to confirm that the skillset produced the expected output.

HTTP

```
### Query the index (indexer status must be "success" before querying the index)
POST {{baseUrl}}/indexes/hotel-reviews-kstore-idxn/docs/search?api-version=2025-09-01
HTTP/1.1
Content-Type: application/json
api-key: {{apiKey}}

{
  "search": "*",
```

```

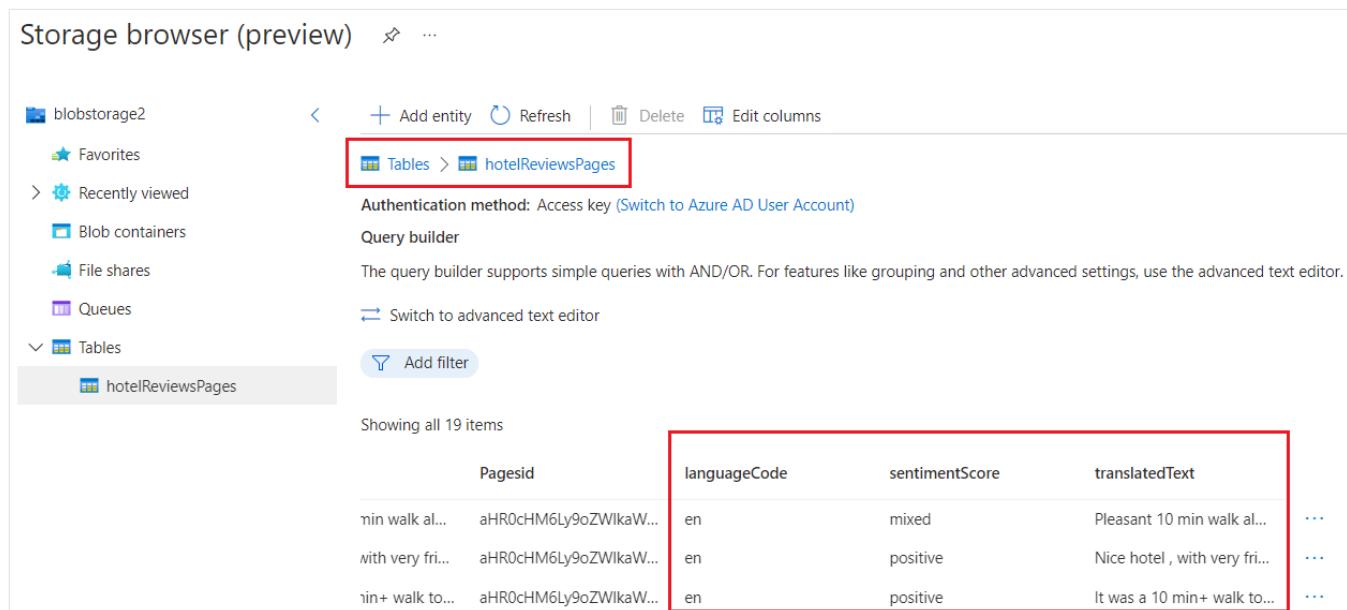
    "select": "reviews_title, reviews_username, language, translated_text,
sentiment",
    "count": true
}

```

Check tables in Azure portal

In the Azure portal, switch to your Azure Storage account and use **Storage Browser** to view the new tables. You should see six tables, one for each projection defined in the skillset.

Each table is generated with the IDs necessary for cross-linking the tables in queries. When you open a table, scroll past these fields to view the content fields added by the pipeline.



The screenshot shows the Azure Storage Explorer interface. On the left, there's a sidebar with icons for blob storage, favorites, recently viewed, blob containers, file shares, queues, and tables. Under 'Tables', 'hotelReviewsPages' is selected. At the top, there are buttons for 'Add entity', 'Refresh', 'Delete', and 'Edit columns'. Below the buttons, the navigation path 'Tables > hotelReviewsPages' is shown, with 'hotelReviewsPages' highlighted by a red box. To the right of the path, it says 'Authentication method: Access key (Switch to Azure AD User Account)'. Below that is a 'Query builder' section with instructions and a link to 'Switch to advanced text editor'. The main area shows a table with 19 items. The columns are 'Pagesid', 'languageCode', 'sentimentScore', and 'translatedText'. The first three rows of data are:

Pagesid	languageCode	sentimentScore	translatedText
aHR0cHM6Ly9oZWlkaw...	en	mixed	Pleasant 10 min walk al...
aHR0cHM6Ly9oZWlkaw...	en	positive	Nice hotel , with very fri...
aHR0cHM6Ly9oZWlkaw...	en	positive	It was a 10 min+ walk to...

In this walkthrough, the knowledge store is composed of a various tables showing different ways of shaping and structuring a table. Tables one through three use output from a Shaper skill to determine the columns and rows. Tables four through six are created from inline shaping instructions, embedded within the projection itself. You can use either approach to achieve the same outcome.

[\[\] Expand table](#)

Table	Description
hotelReviews1Document	Contains fields carried forward from the CSV, such as reviews_date and reviews_text.
hotelReviews2Pages	Contains enriched fields created by the skillset, such as sentiment score and translated text.
hotelReviews3KeyPhrases	Contains a long list of just the key phrases.

Table	Description
hotelReviews4InlineProjectionDocument	Alternative to the first table, using inline shaping instead of the Shaper skill to shape data for the projection.
hotelReviews5InlineProjectionPages	Alternative to the second table, using inline shaping.
hotelreviews6InlineProjectionKeyPhrases	Alternative to the third table, using inline shaping.

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the Azure portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

Next steps

Now that you've enriched your data by using Azure AI services and projected the results to a knowledge store, you can use Storage Explorer or other apps to explore your enriched data set.

[Get started with Storage Explorer](#)

Last updated on 08/21/2025

Shaping data for projection into a knowledge store

! Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in [agentic retrieval](#) workflows.

In Azure AI Search, "shaping data" describes a step in the [knowledge store workflow](#) that creates a data representation of the content that you want to project into tables, objects, and files in Azure Storage.

As skills execute, the outputs are written to an enrichment tree in a hierarchy of nodes, and while you might want to view and consume the enrichment tree in its entirety, it's more likely that you'll want a finer grain, creating subsets of nodes for different scenarios, such as placing the nodes related to translated text or extracted entities in specific tables.

By itself, the enrichment tree doesn't include logic that would inform how its content is represented in a knowledge store. Data shapes fill this gap by providing the schema of what goes into each table, object, and file projection. You can think of a data shape as a custom definition or view of the enriched data. You can create as many shapes as you need, and then assign them to [projections](#) in a knowledge store definition.

Approaches for creating shapes

There are two ways to shape enriched content so that it can be projected into a knowledge store:

- Use the [Shaper skill](#) to create nodes in an enrichment tree that are used expressly for projection. Most skills create new content. In contrast, a Shaper skill works with existing nodes, usually to consolidate multiple nodes into a single complex object. This is useful for tables, where you want the output of multiple nodes to be physically expressed as columns in the table.
- Use an inline shape within the projection definition itself.

Using the Shaper skill externalizes the shape so that it can be used by multiple projections or even other skills. It also ensures that all the mutations of the enrichment tree are contained within the skill, and that the output is an object that can be reused. In contrast, inline shaping

allows you to create the shape you need, but is an anonymous object and is only available to the projection for which it's defined.

The approaches can be used together or separately. This article shows both: a Shaper skill for the table projections, and inline shaping with the key phrases table projection.

Use a Shaper skill

Shaper skills are usually placed at the end of a skillset, creating a view of the data that you want to pass to a projection. This example creates a shape called "tableprojection" containing the following nodes: "reviews_text", "reviews_title", "AzureSearch_DocumentKey", and sentiment scores and key phrases from paged reviews.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "#5",
    "description": null,
    "context": "/document",
    "inputs": [
        {
            "name": "reviews_text",
            "source": "/document/reviews_text",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "reviews_title",
            "source": "/document/reviews_title",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "AzureSearch_DocumentKey",
            "source": "/document/AzureSearch_DocumentKey",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "pages",
            "source": null,
            "sourceContext": "/document/reviews_text/pages/*",
            "inputs": [
                {
                    "name": "Sentiment",
                    "source": "/document/reviews_text/pages/*/Sentiment",
                    "sourceContext": null,
                    "inputs": []
                }
            ],
            "outputs": [
                {
                    "name": "keyPhrases"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "tableprojection"
        }
    ]
}
```

```

    },
    "name": "LanguageCode",
    "source": "/document/Language",
    "sourceContext": null,
    "inputs": []
},
{
    "name": "Page",
    "source": "/document/reviews_text/pages/*",
    "sourceContext": null,
    "inputs": []
},
{
    "name": "keyphrase",
    "sourceContext": "/document/reviews_text/pages/*/Keyphrases/*",
    "inputs": [
        {
            "source": "/document/reviews_text/pages/*/Keyphrases/*",
            "name": "Keyphrases"
        }
    ]
}
],
"outputs": [
    {
        "name": "output",
        "targetName": "tableprojection"
    }
]
}

```

SourceContext property

Within a Shaper skill, an input can have a `sourceContext` element. This same property can also be used in inline shapes in projections.

`sourceContext` is used to construct multi-level, nested objects in an enrichment pipeline. If the input is at a *different* context than the skill context, use the `sourceContext`. The `sourceContext` requires you to define a nested input with the specific element being addressed as the source.

In the previous example, sentiment analysis and key phrases extraction was performed on text that was split into pages for more efficient analysis. Assuming you want the scores and phrases projected into a table, you'll now need to set the context to nested input that provides the score and phrase.

Projecting a shape into multiple tables

With the `tableprojection` node defined in the `outputs` in the previous section, you can slice parts of the `tableprojection` node into individual, related tables:

JSON

```
"projections": [
  {
    "tables": [
      {
        "tableName": "hotelReviewsDocument",
        "generatedKeyName": "Documentid",
        "source": "/document/tableprojection"
      },
      {
        "tableName": "hotelReviewsPages",
        "generatedKeyName": "Pagesid",
        "source": "/document/tableprojection/pages/*"
      },
      {
        "tableName": "hotelReviewsKeyPhrases",
        "generatedKeyName": "KeyPhrasesid",
        "source": "/document/tableprojection/pages/*/keyphrase/*"
      }
    ]
  }
]
```

Inline shape for table projections

Inline shaping is the ability to form new shapes within the projection definition itself. Inline shaping has these characteristics:

- The shape is used only by the projection that contains it.
- The shape can be identical to what a Shaper skill produces.

An inline shape is created using `sourceContext` and `inputs`.

 Expand table

Property	Description
<code>sourceContext</code>	Sets the root of the projection.
<code>inputs</code>	Each input is a column in the table. Name is the column name. Source is the enrichment node that provides the value.

To project the same data as the previous example, the inline projection option would look like this:

JSON

```
"projections": [
  {
    "tables": [
      {
        "tableName": "hotelReviewsInlineDocument",
        "generatedKeyName": "Documentid",
        "sourceContext": "/document",
        "inputs": [
          {
            "name": "reviews_text",
            "source": "/document/reviews_text"
          },
          {
            "name": "reviews_title",
            "source": "/document/reviews_title"
          },
          {
            "name": "AzureSearch_DocumentKey",
            "source": "/document/AzureSearch_DocumentKey"
          }
        ]
      },
      {
        "tableName": "hotelReviewsInlinePages",
        "generatedKeyName": "Pagesid",
        "sourceContext": "/document/reviews_text/pages/*",
        "inputs": [
          {
            "name": "Sentiment",
            "source": "/document/reviews_text/pages/*/Sentiment"
          },
          {
            "name": "LanguageCode",
            "source": "/document/Language"
          },
          {
            "name": "Page",
            "source": "/document/reviews_text/pages/*"
          }
        ]
      },
      {
        "tableName": "hotelReviewsInlineKeyPhrases",
        "generatedKeyName": "KeyPhraseId",
        "sourceContext": "/document/reviews_text/pages/*/Keyphrases/*",
        "inputs": [
          {
            "name": "Keyphrases",
            "source": "/document/reviews_text/pages/*/Keyphrases/*"
          }
        ]
      }
    ]
  }
]
```

```
    }  
]
```

One observation from both the approaches is how values of "Keyphrases" are projected using the "sourceContext". The "Keyphrases" node, which contains a collection of strings, is itself a child of the page text. However, because projections require a JSON object and the page is a primitive (string), the "sourceContext" is used to wrap the key phrase into an object with a named property. This technique enables even primitives to be projected independently.

Inline shape for object projections

You can generate a new shape using the Shaper skill or use inline shaping of the object projection. While the tables example demonstrated the approach of creating a shape and slicing, this example demonstrates the use of inline shaping.

Inline shaping is the ability to create a new shape in the definition of the inputs to a projection. Inline shaping creates an anonymous object that is identical to what a Shaper skill would produce (in this case, `projectionShape`). Inline shaping is useful if you're defining a shape that you don't plan to reuse.

The `projections` property is an array. This example adds a new projection instance to the array, where the `knowledgeStore` definition contains inline projections. When using inline projections, you can omit the Shaper skill.

JSON

```
"knowledgeStore" : {  
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct  
Name>;AccountKey=<Acct Key>;",  
    "projections": [  
        {  
            "tables": [ ],  
            "objects": [  
                {  
                    "storageContainer": "sampleobject",  
                    "source": null,  
                    "generatedKeyName": "myobject",  
                    "sourceContext": "/document",  
                    "inputs": [  
                        {  
                            "name": "metadata_storage_name",  
                            "source": "/document/metadata_storage_name"  
                        },  
                        {  
                            "name": "metadata_storage_path",  
                            "source": "/document/metadata_storage_path"  
                        },  
                        {  
                            "name": "metadata_storage_type",  
                            "source": "/document/metadata_storage_type"  
                        }  
                    ]  
                }  
            ]  
        }  
    ]  
}
```

```
        {
          "name": "content",
          "source": "/document/content"
        },
        {
          "name": "keyPhrases",
          "source": "/document/merged_content/keyphrases/*"
        },
        {
          "name": "entities",
          "source": "/document/merged_content/entities/*/name"
        },
        {
          "name": "ocrText",
          "source": "/document/normalized_images/*/text"
        },
        {
          "name": "ocrLayoutText",
          "source": "/document/normalized_images/*/layoutText"
        }
      ]
    }
  ],
  "files": []
}
]
```

Next steps

This article describes the concepts and principles of projection shapes. As a next step, see how these are applied in patterns for table, object, and file projections.

[Define projections in a knowledge store](#)

Last updated on 10/21/2025

Define projections in a knowledge store

(!) Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in [agentic retrieval](#) workflows.

Projections are the component of a [knowledge store definition](#) that determines how AI enriched content is stored in Azure Storage. Projections determine the type, quantity, and composition of the data structures containing your content.

In this article, learn the syntax for each type of projection:

- [Table projections](#)
- [Object projections](#)
- [File projections](#)

Recall that projections are defined under the `knowledgeStore` property of a skillset.

JSON

```
"knowledgeStore" : {  
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct  
Name>;AccountKey=<Acct Key>;",  
    "projections": [  
        {  
            "tables": [ ],  
            "objects": [ ],  
            "files": [ ]  
        }  
    ]  
}
```

If you need more background before getting started, review [this check list](#) for tips and workflow.

💡 Tip

When developing projections, [enable enrichment caching \(preview\)](#) so that you can reuse existing enrichments while editing projection definitions. Enrichment caching is a preview feature, so be sure to use the preview REST API on the indexer request. Without caching, simple edits to a projection will result in a full reprocess of enriched content. By caching

the enrichments, you can iterate over projections without incurring any skillset processing charges.

Requirements

All projections have source and destination properties. The source is always internal content from an enrichment tree created during skillset execution. The destination is the name and type of an external object that's created and populated in Azure Storage.

Except for file projections, which only accept binary images, the source must be:

- Valid JSON
- A path to a node in the enrichment tree (for example, `"source": "/document/objectprojection"`)

While a node might resolve to a single field, a more common representation is a reference to a complex shape. Complex shapes are created through a shaping methodology, either a [Shaper skill](#) or an [inline shaping definition](#), but usually a Shaper skill. The fields or elements of the shape determine the fields in containers and tables.

Shaper skills are favored because it outputs JSON, whereas most skills don't output valid JSON on their own. In many cases, the same data shape created by a Shaper skill can be used equally by both table and object projections.

Given source input requirements, knowing how to [shape data](#) becomes a practical requirement for projection definition, especially if you're working with tables.

Define a table projection

Table projections are recommended for scenarios that call for data exploration, such as analysis with Power BI or workloads that consume data frames. The tables section of a projections array is a list of tables that you want to project.

To define a table projection, use the `tables` array in the `projections` property. A table projection has three required properties:

 [Expand table](#)

Property	Description
tableName	Determines the name of a new table created in Azure Table Storage.

Property	Description
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column is created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree. The node should be a reference to a complex shape that determines which columns are created in the table.

In table projections, "source" is usually the output of a [Shaper skill](#) that defines the shape of the table. Tables have rows and columns, and shaping is the mechanism by which rows and columns are specified. You can use a [Shaper skill or inline shapes](#). The Shaper skill produces valid JSON, but the source could be the output from any skill, if valid JSON.

(!) Note

Table projections are subject to the [storage limits](#) imposed by Azure Storage. The entity size can't exceed 1 MB and a single property can be no bigger than 64 KB. These constraints make tables a good solution for storing a large number of small entities.

Single table example

The schema of a table is specified partly by the projection (table name and key), and also by the source that provides the shape of table (columns). This example shows just one table so that you can focus on the details of the definition.

JSON

```
"projections" : [
  {
    "tables": [
      { "tableName": "Hotels", "generatedKeyName": "HotelId", "source":
"/document/tableprojection" }
    ]
  }
]
```

Columns are derived from the "source". The following data shape containing HotelId, HotelName, Category, and Description will result in creation of those columns in the table.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "#3",
```

```

"description": null,
"context": "/document",
"inputs": [
{
  "name": "HotelId",
  "source": "/document/HotelId"
},
{
  "name": "HotelName",
  "source": "/document/HotelName"
},
{
  "name": "Category",
  "source": "/document/Category"
},
{
  "name": "Description",
  "source": "/document/Description"
},
],
"outputs": [
{
  "name": "output",
  "targetName": "tableprojection"
}
]
}

```

Multiple table (slicing) example

A common pattern for table projections is to have multiple related tables, where system-generated partitionKey and rowKey columns are created to support cross-table relationships for all tables under the same projection group.

Creating multiple tables can be useful if you want control over how related data is aggregated. If enriched content has unrelated or independent components, for example the keywords extracted from a document might be unrelated from the entities recognized in the same document, you can split out those fields into adjacent tables.

When you're projecting to multiple tables, the complete shape is projected into each table, unless a child node is the source of another table within the same group. Adding a projection with a source path that is a child of an existing projection results in the child node being sliced out of the parent node and projected into the new yet related table. This technique allows you to define a single node in a Shaper skill that can be the source for all of your projections.

The pattern for multiple tables consists of:

- One table as the parent or main table
- Other tables to contain slices of the enriched content

For example, assume a Shaper skill outputs an "EnrichedShape" that contains hotel information, plus enriched content like key phrases, locations, and organizations. The main table would include fields that describe the hotel (ID, name, description, address, category). Key phrases would get the key phrase column. Entities would get the entity columns.

JSON

```
"projections" : [
  {
    "tables": [
      { "tableName": "MainTable", "generatedKeyName": "HotelId", "source": "/document/EnrichedShape" },
      { "tableName": "KeyPhrases", "generatedKeyName": "KeyPhraseId", "source": "/document/EnrichedShape/*/KeyPhrases/*" },
      { "tableName": "Entities", "generatedKeyName": "EntityId", "source": "/document/EnrichedShape/*/Entities/*" }
    ]
  }
]
```

Naming relationships

The `generatedKeyName` and `referenceKeyName` properties are used to relate data across tables or even across projection types. Each row in the child table has a property pointing back to the parent. The name of the column or property in the child is the `referenceKeyName` from the parent. When the `referenceKeyName` isn't provided, the service defaults it to the `generatedKeyName` from the parent.

Power BI relies on these generated keys to discover relationships within the tables. If you need the column in the child table named differently, set the `referenceKeyName` property on the parent table. One example would be to set the `generatedKeyName` as ID on the `tblDocument` table and the `referenceKeyName` as `DocumentID`. This would result in the column in the `tblEntities` and `tblKeyPhrases` tables containing the document ID being named `DocumentID`.

Define an object projection

Object projections are JSON representations of the enrichment tree that can be sourced from any node. In comparison with table projections, object projections are simpler to define and are used when projecting whole documents. Object projections are limited to a single projection in a container and can't be sliced.

To define an object projection, use the `objects` array in the `projections` property. An object projection has three required properties:

Property	Description
storageContainer	Determines the name of a new container created in Azure Storage.
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column is created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree that is the root of the projection. The node is usually a reference to a complex data shape that determines blob structure.

The following example projects individual hotel documents, one hotel document per blob, into a container called `hotels`.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "an Azure storage connection string",
  "projections" : [
    {
      "tables": [ ]
    },
    {
      "objects": [
        {
          "storageContainer": "hotels",
          "source": "/document/objectprojection",
        }
      ]
    },
    {
      "files": [ ]
    }
  ]
}
```

The source is the output of a Shaper skill, named `"objectprojection"`. Each blob has a JSON representation of each field input.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "#3",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "HotelId",
      "type": "Edm.Int32"
    }
  ],
  "outputs": [
    {
      "name": "Hotel"
    }
  ],
  "constraints": [
    {
      "name": "HotelId"
    }
  ],
  "skills": [
    {
      "name": "Hotel"
    }
  ],
  "parameters": [
    {
      "name": "HotelId"
    }
  ],
  "version": "1.0"
}
```

```

        "source": "/document/HotelId"
    },
    {
        "name": "HotelName",
        "source": "/document/HotelName"
    },
    {
        "name": "Category",
        "source": "/document/Category"
    },
    {
        "name": "keyPhrases",
        "source": "/document/HotelId/keyphrases/*"
    },
],
"outputs": [
{
    "name": "output",
    "targetName": "objectprojection"
}
]
}
}

```

Define a file projection

File projections are always binary, normalized images, where normalization refers to potential resizing and rotation for use in skillset execution. File projections, similar to object projections, are created as blobs in Azure Storage, and contain binary data (as opposed to JSON).

To define a file projection, use the `files` array in the `projections` property. A files projection has three required properties:

[Expand table](#)

Property	Description
storageContainer	Determines the name of a new container created in Azure Storage.
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column is created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree that is the root of the projection. For images files, the source is always <code>/document/normalized_images/*</code> . File projections only act on the <code>normalized_images</code> collection. Neither indexers nor a skillset will pass through the original non-normalized image.

The destination is always a blob container, with a folder prefix of the base64 encoded value of the document ID. If there are multiple images, they're placed together in the same folder. File

projections can't share the same container as object projections and need to be projected into a different container.

The following example projects all normalized images extracted from the document node of an enriched document, into a container called `myImages`.

JSON

```
"projections": [
  {
    "tables": [ ],
    "objects": [ ],
    "files": [
      {
        "storageContainer": "myImages",
        "source": "/document/normalized_images/*"
      }
    ]
  }
]
```

Test projections

You can process projections by following these steps:

1. Set the knowledge store's `storageConnectionString` property to a valid V2 general purpose storage account connection string.
2. [Update the skillset](#) by issuing a PUT request with your projection definition in the body of the skillset.
3. [Run the indexer](#) to put the skillset into execution.
4. [Monitor indexer execution](#) to check progress and catch any errors.
5. Use Azure portal to verify object creation in Azure Storage.
6. If you're projecting tables, [import them into Power BI](#) for table manipulation and visualization. In most cases, Power BI autodiscovers the relationships among tables.

Common issues

Omitting any of the following steps can result in unexpected outcomes. Check for the following conditions if your output doesn't look right.

- String enrichments aren't shaped into valid JSON. When strings are enriched, for example `merged_content` enriched with key phrases, the enriched property is represented as a child of `merged_content` within the enrichment tree. The default representation isn't well-formed JSON. At projection time, make sure to transform the enrichment into a valid JSON object with a name and a value. Using a Shaper skill or defining inline shapes help resolve this issue.
- Omission of `/*` at the end of a source path. If the source of a projection is `/document/projectionShape/keyPhrases`, the key phrases array is projected as a single object/row. Instead, set the source path to `/document/projectionShape/keyPhrases/*` to yield a single row or object for each of the key phrases.
- Path syntax errors. [Path selectors](#) are case-sensitive and can lead to missing input warnings if you don't use the exact case for the selector.

Next steps

The next step walks you through shaping and projection of output from a rich skillset. If your skillset is complex, the following article provides examples of both shapes and projections.

[Detailed example of shapes and projections](#)

Last updated on 10/21/2025

Example of shapes and projections in a knowledge store

! Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in [agentic retrieval](#) workflows.

This article provides a detailed example that supplements [high-level concepts](#) and [syntax-based articles](#) by walking you through the shaping and projection steps required for fully expressing the output of a rich skillset in a [knowledge store](#) in Azure Storage.

If your application requirements call for multiple skills and projections, this example can give you a better idea of how shapes and projections interact.

Set up sample data

Sample documents aren't included with the Projections collection, but the [AI enrichment demo data files](#) ↗ contain text and images that work with the projections described in this example. If you use this sample data, you can skip step that [attaches a Microsoft Foundry resource](#) because you stay under the daily indexer limit for free enrichments.

Create a blob container in Azure Storage and upload all 14 items.

While in Azure Storage, copy a connection string.

You can use the [projections.rest](#) ↗ file to run the examples in this article.

Example skillset

To understand the dependency between shapes and projections, review the following skillset that creates enriched content. This skillset processes both raw images and text, producing outputs that will be referenced in shapes and projections.

Pay close attention to skill outputs (targetNames). Outputs written to the enriched document tree are referenced in projections and in shapes (via Shaper skills).

JSON

```
{
  "name": "projections-demo-ss",
  "description": "Skillset that enriches blob data found in the merged_content field. The enrichment granularity is a document.",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "name": "#1",
      "description": null,
      "context": "/document/merged_content",
      "categories": [
        "Person",
        "Quantity",
        "Organization",
        "URL",
        "Email",
        "Location",
        "DateTime"
      ],
      "defaultLanguageCode": "en",
      "minimumPrecision": null,
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_content"
        },
        {
          "name": "languageCode",
          "source": "/document/language"
        }
      ],
      "outputs": [
        {
          "name": "persons",
          "targetName": "people"
        },
        {
          "name": "organizations",
          "targetName": "organizations"
        },
        {
          "name": "locations",
          "targetName": "locations"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
      "name": "#2",
      "description": null,
      "context": "/document/merged_content",
      "defaultLanguageCode": "en",
      "maxKeyPhraseCount": null,
      "inputs": [
        {
          "name": "keyPhrases"
        }
      ]
    }
  ]
}
```

```
{
    "name": "text",
    "source": "/document/merged_content"
},
{
    "name": "languageCode",
    "source": "/document/language"
}
],
"outputs": [
{
    "name": "keyPhrases",
    "targetName": "keyphrases"
}
]
},
{
    "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
    "name": "#3",
    "description": null,
    "context": "/document",
    "inputs": [
{
    "name": "text",
    "source": "/document/merged_content"
}
],
"outputs": [
{
    "name": "languageCode",
    "targetName": "language"
}
]
},
{
    "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
    "name": "#4",
    "description": null,
    "context": "/document",
    "insertPreTag": " ",
    "insertPostTag": " ",
    "inputs": [
{
    "name": "text",
    "source": "/document/content"
},
{
    "name": "itemsToInsert",
    "source": "/document/normalized_images/*/text"
},
{
    "name": "offsets",
    "source": "/document/normalized_images/*/contentOffset"
}
],
},
```

```

    "outputs": [
      {
        "name": "mergedText",
        "targetName": "merged_content"
      }
    ]
  },
  {
    "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
    "name": "#5",
    "description": null,
    "context": "/document/normalized_images/*",
    "textExtractionAlgorithm": "printed",
    "lineEnding": "Space",
    "defaultLanguageCode": "en",
    "detectOrientation": true,
    "inputs": [
      {
        "name": "image",
        "source": "/document/normalized_images/*"
      }
    ],
    "outputs": [
      {
        "name": "text",
        "targetName": "text"
      },
      {
        "name": "layoutText",
        "targetName": "layoutText"
      }
    ]
  }
],
"cognitiveServices": {
  "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",
  "description": "A Foundry resource in the same region as Search.",
  "key": ""
},
"knowledgeStore": null
}

```

Note

Under `"cognitiveServices"`, the key field is unspecified because the indexer can use a Foundry resource in the same region as your search service and process up to 20 transactions daily at no charge. The sample data for this example stays under the 20 transaction limit.

Example Shaper skill

A [Shaper skill](#) is a utility for working with existing enriched content instead of creating new enriched content. Adding a Shaper to a skillset lets you create a custom shape that you can project into table or blob storage. Without a custom shape, projections are limited to referencing a single node (one projection per output), which isn't suitable for tables. Creating a custom shape aggregates various elements into a new logical whole that can be projected as a single table, or sliced and distributed across a collection of tables.

In this example, the custom shape combines blob metadata and identified entities and key phrases. The custom shape is called `projectionShape` and is parented under `/document`.

One purpose of shaping is to ensure that all enrichment nodes are expressed in well-formed JSON, which is required for projecting into knowledge store. This is especially true when an enrichment tree contains nodes that aren't well-formed JSON (for example, when an enrichment is parented to a primitive like a string).

Notice the last two nodes, `KeyPhrases` and `Entities`. These are wrapped into a valid JSON object with the `sourceContext`. This is required as `keyphrases` and `entities` are enrichments on primitives and need to be converted to valid JSON before they can be projected.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "ShaperForTables",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "metadata_storage_content_type",
      "source": "/document/metadata_storage_content_type",
      "sourceContext": null,
      "inputs": []
    },
    {
      "name": "metadata_storage_name",
      "source": "/document/metadata_storage_name",
      "sourceContext": null,
      "inputs": []
    },
    {
      "name": "metadata_storage_path",
      "source": "/document/metadata_storage_path",
      "sourceContext": null,
      "inputs": []
    },
    {
      "name": "keyphrases",
      "source": "/document/keyphrases",
      "sourceContext": {
        "key": "keyphrases"
      },
      "inputs": []
    },
    {
      "name": "entities",
      "source": "/document/entities",
      "sourceContext": {
        "key": "entities"
      },
      "inputs": []
    }
  ]
}
```

```

        "name": "metadata_content_type",
        "source": "/document/metadata_content_type",
        "sourceContext": null,
        "inputs": []
    },
    {
        "name": "keyPhrases",
        "source": null,
        "sourceContext": "/document/merged_content/keyphrases/*",
        "inputs": [
            {
                "name": "KeyPhrases",
                "source": "/document/merged_content/keyphrases/*"
            }
        ]
    },
    {
        "name": "Entities",
        "source": null,
        "sourceContext": "/document/merged_content/entities/*",
        "inputs": [
            {
                "name": "Entities",
                "source": "/document/merged_content/entities/*/name"
            }
        ]
    }
],
"outputs": [
    {
        "name": "output",
        "targetName": "projectionShape"
    }
]
}

```

Add Shapers to a skillset

The example skillset introduced at the start of this article didn't include the Shaper skill, but Shaper skills belong in a skillset and are often placed towards the end.

Within a skillset, a Shaper skill might look like this:

JSON

```
{
    "name": "projections-demo-ss",
    "skills": [
        {
            <Shaper skill goes here>
        }
    ]
}
```

```

        }
    ],
    "cognitiveServices": "A key goes here",
    "knowledgeStore": []
}

```

Projecting to tables

Drawing on the examples above, there's a known quantity of enrichments and data shapes that can be referenced in table projections. In the tables projection below, three tables are defined by setting the `tableName`, `source` and `generatedKeyName` properties.

All three of these tables will be related through generated keys and by the shared parent `/document/projectionShape`.

JSON

```

"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct
Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "tblDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/projectionShape"
                },
                {
                    "tableName": "tblKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/projectionShape/keyPhrases/*"
                },
                {
                    "tableName": "tblEntities",
                    "generatedKeyName": "Entityid",
                    "source": "/document/projectionShape/Entities/*"
                }
            ],
            "objects": [],
            "files": []
        }
    ]
}

```

Test your work

You can check projection definitions by following these steps:

1. Set the knowledge store's `storageConnectionString` property to a valid V2 general purpose storage account connection string.
2. Update the skillset by issuing the PUT request.
3. After updating the skillset, run the indexer.

You now have a working projection with three tables. [Importing these tables into Power BI](#) should result in Power BI discovering the relationships.

Before moving on to the next example, let's revisit aspects of the table projection to understand the mechanics of slicing and relating data.

Slicing a table into multiple child tables

Slicing is a technique that subdivides a whole consolidated shape into constituent parts. The outcome consists of separate but related tables that you can work with individually.

In the example, `projectionShape` is the consolidated shape (or enrichment node). In the projection definition, `projectionShape` is sliced into additional tables, which enables you to pull out parts of the shape, `keyPhrases` and `Entities`. In Power BI, this is useful as multiple entities and keyPhrases are associated with each document, and you'll get more insights if you can see entities and keyPhrases as categorized data.

Slicing implicitly generates a relationship between the parent and child tables, using the `generatedKeyName` in the parent table to create a column with the same name in the child table.

Naming relationships

The `generatedKeyName` and `referenceKeyName` properties are used to relate data across tables or even across projection types. Each row in the child table has a property pointing back to the parent. The name of the column or property in the child is the `referenceKeyName` from the parent. When the `referenceKeyName` isn't provided, the service defaults it to the `generatedKeyName` from the parent.

Power BI relies on these generated keys to discover relationships within the tables. If you need the column in the child table named differently, set the `referenceKeyName` property on the parent table. One example would be to set the `generatedKeyName` as ID on the `tblDocument` table and the `referenceKeyName` as `DocumentID`. This would result in the column in the `tblEntities` and `tblKeyPhrases` tables containing the document ID being named `DocumentID`.

Projecting blob documents

Object projections are JSON representations of the enrichment tree that can be sourced from any node. In comparison with table projections, object projections are simpler to define and are used when projecting whole documents. Object projections are limited to a single projection in a container and can't be sliced.

To define an object projection, use the `objects` array in the `projections` property.

The source is the path to a node of the enrichment tree that is the root of the projection. Although it isn't required, the node path is usually the output of a Shaper skill. This is because most skills don't output valid JSON objects on their own, which means that some form of shaping is necessary. In many cases, the same Shaper skill that creates a table projection can be used to generate an object projection. Alternatively, the source can also be set to a node with [an inline shaping](#) to provide the structure.

The destination is always a blob container.

The following example projects individual hotel documents, one hotel document per blob, into a container called `hotels`.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "an Azure storage connection string",
  "projections" : [
    {
      "tables": [ ]
    },
    {
      "objects": [
        {
          "storageContainer": "hotels",
          "source": "/document/objectprojection",
        }
      ]
    },
    {
      "files": [ ]
    }
  ]
}
```

The source is the output of a Shaper skill, named `"objectprojection"`. Each blob will have a JSON representation of each field input.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "#3",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "HotelId",
      "source": "/document/HotelId"
    },
    {
      "name": "HotelName",
      "source": "/document/HotelName"
    },
    {
      "name": "Category",
      "source": "/document/Category"
    },
    {
      "name": "keyPhrases",
      "source": "/document/HotelId/keyphrases/*"
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "objectprojection"
    }
  ]
}
```

Projecting an image file

File projections are always binary, normalized images, where normalization refers to potential resizing and rotation for use in skillset execution. File projections, similar to object projections, are created as blobs in Azure Storage, and contain the image.

To define a file projection, use the `files` array in the `projections` property.

The source is always `/document/normalized_images/*`. File projections only act on the `normalized_images` collection. Neither indexers nor a skillset will pass through the original non-normalized image.

The destination is always a blob container, with a folder prefix of the base64 encoded value of the document ID. File projections can't share the same container as object projections and need to be projected into a different container.

The following example projects all normalized images extracted from the document node of an enriched document, into a container called `myImages`.

JSON

```
"knowledgeStore" : {
  "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct
Name>;AccountKey=<Acct Key>;",
  "projections": [
    {
      "tables": [ ],
      "objects": [ ],
      "files": [
        {
          "storageContainer": "myImages",
          "source": "/document/normalized_images/*"
        }
      ]
    }
  ]
}
```

Projecting to multiple types

A more complex scenario might require you to project content across projection types. For example, projecting key phrases and entities to tables, saving OCR results of text and layout text as objects, and then projecting the images as files.

Steps for multiple projection types:

1. Create a table with a row for each document.
2. Create a table related to the document table with each key phrase identified as a row in this table.
3. Create a table related to the document table with each entity identified as a row in this table.
4. Create an object projection with the layout text for each image.
5. Create a file projection, projecting each extracted image.
6. Create a cross-reference table that contains references to the document table, object projection with the layout text, and the file projection.

Shape data for cross-projection

To get the shapes needed for these projections, start by adding a new Shaper skill that creates a shaped object called `crossProjection`.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "ShaperForCrossProjection",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "metadata_storage_name",
      "source": "/document/metadata_storage_name",
      "sourceContext": null,
      "inputs": []
    },
    {
      "name": "keyPhrases",
      "source": null,
      "sourceContext": "/document/merged_content/keyphrases/*",
      "inputs": [
        {
          "name": "KeyPhrases",
          "source": "/document/merged_content/keyphrases/*"
        }
      ]
    },
    {
      "name": "entities",
      "source": null,
      "sourceContext": "/document/merged_content/entities/*",
      "inputs": [
        {
          "name": "Entities",
          "source": "/document/merged_content/entities/*/name"
        }
      ]
    },
    {
      "name": "images",
      "source": null,
      "sourceContext": "/document/normalized_images/*",
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        },
        {
          "name": "layoutText",
          "source": "/document/normalized_images/*/layoutText"
        },
        {
          "name": "ocrText",
          "source": "/document/normalized_images/*/text"
        }
      ]
    }
  ]
}
```

```

        ],
        "outputs": [
            {
                "name": "output",
                "targetName": "crossProjection"
            }
        ]
    }
}

```

Define table, object, and file projections

From the consolidated crossProjection object, slice the object into multiple tables, capture the OCR output as blobs, and then save the image as files (also in Blob storage).

JSON

```

"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct
Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "crossDocument",
                    "generatedKeyName": "Id",
                    "source": "/document/crossProjection"
                },
                {
                    "tableName": "crossEntities",
                    "generatedKeyName": "EntityId",
                    "source": "/document/crossProjection/entities/*"
                },
                {
                    "tableName": "crossKeyPhrases",
                    "generatedKeyName": "KeyPhraseId",
                    "source": "/document/crossProjection/keyPhrases/*"
                },
                {
                    "tableName": "crossReference",
                    "generatedKeyName": "CrossId",
                    "source": "/document/crossProjection/images/*"
                }
            ],
            "objects": [
                {
                    "storageContainer": "crossobject",
                    "generatedKeyName": "crosslayout",
                    "source": null,

```

```

        "sourceContext":  

        "/document/crossProjection/images/*/layoutText",  

        "inputs": [  

            {  

                "name": "OcrLayoutText",  

                "source":  

                "/document/crossProjection/images/*/layoutText"  

            }  

        ]  

    },  

    "files": [  

        {  

            "storageContainer": "crossimages",  

            "generatedKeyName": "crossimages",  

            "source": "/document/crossProjection/images/*/image"  

        }  

    ]  

}  

]
}

```

Object projections require a container name for each projection. Object projections and file projections can't share a container.

Relationships among table, object, and file projections

This example also highlights another feature of projections. By defining multiple types of projections within the same projection object, there's a relationship expressed within and across the different types (tables, objects, files). This allows you to start with a table row for a document and find all the OCR text for the images within that document in the object projection.

If you don't want the data related, define the projections in different projection groups. For example, the following snippet will result in the tables being related, but without relationships between the tables and the object (OCR text) projections.

Projection groups are useful when you want to project the same data in different shapes for different needs. For example, a projection group for the Power BI dashboard, and another projection group for capturing data used to train a machine learning model wrapped in a custom skill.

When building projections of different types, file and object projections are generated first, and the paths are added to the tables.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct
Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "unrelatedDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/projectionShape"
                },
                {
                    "tableName": "unrelatedKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/projectionShape/keyPhrases"
                }
            ],
            "objects": [
                ],
            "files": []
        },
        {
            "tables": [],
            "objects": [
                {
                    "storageContainer": "unrelatedocrtext",
                    "source": null,
                    "sourceContext": "/document/normalized_images/*/text",
                    "inputs": [
                        {
                            "name": "ocrText",
                            "source": "/document/normalized_images/*/text"
                        }
                    ]
                },
                {
                    "storageContainer": "unrelatedocrlayout",
                    "source": null,
                    "sourceContext": "/document/normalized_images/*/layoutText",
                    "inputs": [
                        {
                            "name": "ocrLayoutText",
                            "source": "/document/normalized_images/*/layoutText"
                        }
                    ]
                }
            ],
            "files": []
        }
    ]
}
```

Next steps

The example in this article demonstrates common patterns on how to create projections. Now that you have a good understanding of the concepts, you're better equipped to build projections for your specific scenario.

[Configure caching for incremental enrichment](#)

Last updated on 10/21/2025

Connect a knowledge store with Power BI

Note

Knowledge stores are secondary storage that exists in Azure Storage and contain the outputs of Azure AI Search skillsets. They're separate from knowledge sources and knowledge bases, which are used in [agentic retrieval](#) workflows.

In this article, learn how to connect to and query a knowledge store using Power Query in the Power BI Desktop app. You can get started faster with templates, or build a custom dashboard from scratch.

A knowledge store that's composed of tables in Azure Storage work best in Power BI. If the tables contain projections from the same skillset and projection group, you can easily "join" them to build table visualizations that include fields from related tables.

Follow the steps in this article using sample data and a knowledge store as [created in this portal quickstart](#) or through [REST APIs](#).

Connect to Azure Storage

1. Start [Power BI Desktop](#) and select **Get data**.
2. In **Get Data**, select **Azure**, and then select **Azure Table Storage**.
3. Select **Connect**.
4. For **Account Name or URL**, enter in your Azure Storage account name (the full URL is created for you).
5. If prompted, enter the storage account key.

Set up tables

1. Select the checkbox next to all of the tables that were created from the same skillset, and then select **Load**.



2. On the top ribbon, select **Transform Data** to open the **Power Query Editor**.



3. Open *hotelReviewsDocument* and remove its *PartitionKey*, *RowKey*, and *Timestamp* columns. Those columns are used for table relationships in Azure Table Storage. Power BI doesn't need them. You should be left with one column named "Content" showing *Record* in each one.



4. Select the icon with opposing arrows at the upper right side of the table to expand *Content*. When the list of columns appears, select all columns. Clear columns starting with 'metadata'. Select **OK** to include the selected columns.

A screenshot of the Microsoft Power Query Editor interface. The title bar says "Untitled - Power Query Editor". The ribbon has tabs: File, Home, Transform, Add Column, View, and Help. The "Transform" tab is selected. On the left, there's a list of columns under "Search Columns to Expand": latitude, longitude, metadata_storage_content_type, metadata_storage_last_modified, metadata_storage_name, metadata_storage_path, metadata_storage_size, name, postalCode, province, reviews_date. A red box highlights the first four items. Below this list is a checked checkbox for "Use original column name as prefix". At the bottom left is a warning message: "⚠ List may be incomplete." with a "Load more" link, and two buttons: "OK" (highlighted with a red box) and "Cancel". On the right, there's a preview pane showing a table with four rows, each labeled "Record". The first row is highlighted with a yellow background. Above the preview are icons for "Data source settings", "Manage Parameters", "Refresh Preview", and "Parameters".

Content
1 Record
2 Record
3 Record
4 Record

5. Change the data type for the following columns by clicking the ABC-123 icon at the top left of the column.

- For *content.latitude* and *Content.longitude*, select **Decimal Number**.
- For *Content.reviews_date* and *Content.reviews_dateAdded*, select **Date/Time**.



	ABC 123	Content.latitude	ABC 123	Content.longitude
1	1.2	Decimal Number	5187	
2	\$	Fixed decimal number	5187	
3	1 ² 3	Whole Number	5187	
4	%	Percentage	5187	
5		Date/Time	5187	
6		Date	5187	
7			5187	

6. Open *hotelReviewsSsPages* and repeat column deletion steps, expanding *Content* to select columns from the records. There are no data type modifications for this table.
7. Open *hotelReviewsSsKeyPhrases* and repeat column deletion steps, expanding *Content* to select columns from the records. There are no data type modifications for this table.
8. On the command bar, select **Close and Apply**.

Check table relationships

1. Select on the Model tile on the left pane and validate that Power BI shows relationships between all three tables.
 Validate relationships
2. Double-click each relationship and make sure that the **Cross-filter direction** is set to **Both**. This enables your visuals to refresh when a filter is applied.

Build a report

1. Select on the Report tile on the left pane to explore data through visualizations. For text fields, tables and cards are useful visualizations.
2. Choose fields from each of the three tables to fill in the table or card.



Build a table report

Sample Power BI template - Azure portal only

When creating a [knowledge store using the Azure portal](#), you have the option of downloading a [Power BI template](#) on the second page of the **Import data** wizard. This template gives you several visualizations, such as WordCloud and Network Navigator, for text-based content.

Select **Get Power BI Template** on the **Add cognitive skills** page to retrieve and download the template from its public GitHub location. The wizard modifies the template to accommodate the shape of your data, as captured in the knowledge store projections specified in the wizard. For this reason, the template you download varies each time you run the wizard, assuming different data inputs and skill selections.



Note

The template is downloaded while the wizard is in mid-flight. You'll have to wait until the knowledge store is actually created in Azure Table Storage before you can use it.

Video introduction

For a demonstration of using Power BI with a knowledge store, watch the following video.

<https://www.youtube-nocookie.com/embed/XWzLBP8iWqg?version=3>

Next steps

[Tables in Power BI reports and dashboards](#)

C# samples for Azure AI Search

09/23/2025

Learn about C# code samples that demonstrate the functionality and workflow of an Azure AI Search solution. These samples use the [Azure AI Search client library](#) for the [Azure SDK for .NET](#), which you can explore through the following links.

[] [Expand table](#)

Target	Link
Package download	nuget.org/packages/Azure.Search.Documents/ ↗
API reference	Azure.Search.Documents
API test cases	github.com/Azure/azure-sdk-for-net/tree/main/sdk/search/Azure.Search.Documents/tests ↗
Source code	github.com/Azure/azure-sdk-for-net/tree/main/sdk/search/Azure.Search/Documents/src ↗
Change log	github.com/Azure/azure-sdk-for-net/blob/main/sdk/search/Azure.Search/Documents/CHANGELOG.md ↗

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in [Azure/azure-sdk-for-net/blob/main/sdk/search/Azure.Search/Documents/samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/search/Azure.Search/Documents/samples) ↗ on GitHub.

[] [Expand table](#)

Sample	Description
Hello world (synchronous) ↗	Create a client, authenticate, and handle errors using synchronous methods.
Hello world (asynchronous) ↗	Create a client, authenticate, and handle errors using asynchronous methods.
Service-level operations ↗	Get service statistics and create multiple search objects, including an index, indexer, data source, skillset, and synonym map. Finally, you query the index.
Index operations ↗	Get a count of documents stored in an index.

Sample	Description
FieldBuilderIgnore ↗	Use an attribute to work with unsupported data types.
Indexing documents (push model) ↗	Use the push model to index documents by sending a JSON payload to an index.
Customer-managed encryption keys ↗	Use a customer-managed encryption key to protect sensitive content.
Vector search ↗	Index a vector field and perform vector search.
Semantic ranking ↗	Configure semantic ranker in an index and run semantic queries.

Doc samples

Code samples from the Azure AI Search team demonstrate features and workflows. The following samples are referenced in tutorials, quickstarts, and how-to articles that explain the code in detail. You can find these samples in [Azure-Samples/azure-search-dotnet-samples ↗](#) and [Azure-Samples/search-dotnet-getting-started ↗](#) on GitHub.

[] [Expand table](#)

Sample	Article	Description
quickstart ↗	Quickstart: Full-text search	Create, load, and query an index using sample data.
quickstart-agentic-retrieval ↗	Quickstart: Agentic retrieval	Integrate semantic ranking with LLM-powered query planning and answer generation.
quickstart-rag ↗	Quickstart: Generative search (RAG)	Use grounding data from Azure AI Search with a chat completion model from Azure OpenAI.
quickstart-semantic-search ↗	Quickstart: Semantic ranking	Add semantic ranking to an index schema and run semantic queries.
quickstart-vector-search ↗	Quickstart: Vector search	Index and query vector content.
create-mvc-app ↗	Tutorial: Add search to an ASP.NET Core (MVC) app	Add basic search, pagination, and other server-side behaviors to an MVC web app, unlike the console applications used in most samples.

Sample	Article	Description
search-website	Tutorial: Add search to web apps	Build an end-to-end search app that uses the push API for bulk upload and a rich client for hosting the app and handling search requests.
tutorial-ai-enrichment	Tutorial: AI-generated searchable content from Azure blobs	Create a skillset that iterates over Azure blobs to extract information and infer structure.
multiple-data-sources	Tutorial: Index from multiple data sources	Merge content from two data sources into one index.
optimize-data-indexing	Tutorial: Optimize indexing with the push API	Use optimization techniques for pushing data into an index.
DotNetHowTo	Use the .NET client library	Create and manage multiple search objects while learning about the APIs.
DotNetToIndexers	Tutorial: Index Azure SQL data	Configure an Azure SQL indexer with a schedule, field mappings, and parameters.
DotNetHowToEncryptionUsingCMK	Configure customer-managed keys for data encryption	Create objects that are encrypted with a customer-managed key.

Accelerators

An accelerator is an end-to-end solution that includes code and documentation you can adapt for your own implementation of a specific scenario.

[Expand table](#)

Sample	Description
search-qna-maker-accelerator	Solution that combines Azure AI Search and QnA Maker. See the live demo site.

Demos

A demo repo provides proof-of-concept source code for examples or scenarios shown in demonstrations. Unlike accelerators, demo solutions aren't designed for adaptation.

[] Expand table

Sample	Description
covid19search ↗	Source code repo for the Azure AI Search-based Covid-19 search app.
AzureSearch_JFK_Files ↗	Source code repo for the Azure AI Search-based JFK files solution.

Other samples

The following samples are also published by the Azure AI Search team but aren't referenced in documentation. Associated README files provide usage instructions.

[] Expand table

Sample	Description
check-storage-usage ↗	Check search service storage on a schedule using an Azure function.
export-data ↗	Partition and export a large index using a C# console app.
index-backup-restore ↗	Copy an index from one service to another, creating JSON files with the index schema and documents.
data-lake-gen2-acl-indexing ↗	Index Azure Data Lake Gen2 files and folders secured with Microsoft Entra ID and role-based access control.
multiple-search-services ↗	Query multiple search services and combine results into a single page.
search-aggregations ↗	Obtain and filter aggregations from an index.
azure-search-power-skills ↗	Incorporate consumable custom skills into your own solutions.
DotNetVectorDemo ↗	Create, load, and query a vector index.
DotNetIntegratedVectorizationDemo ↗	Extend the vector workflow to include skills-based automation for data chunking and embedding.

Tip

Use the [samples browser](#) to search for Microsoft code samples on GitHub. You can filter your search by product, service, and language.

Java samples for Azure AI Search

09/23/2025

Learn about Java code samples that demonstrate the functionality and workflow of an Azure AI Search solution. These samples use the [Azure AI Search client library](#) for the [Azure SDK for Java](#), which you can explore through the following links.

[] [Expand table](#)

Target	Link
Package download	search.maven.org/artifact/com.azure/azure-search-documents ↗
API reference	com.azure.search.documents
API test cases	github.com/Azure/azure-sdk-for-java/tree/main/sdk/search/azure-search-documents/src/test ↗
Source code	github.com/Azure/azure-sdk-for-java/tree/main/sdk/search/azure-search-documents ↗
Change log	github.com/Azure/azure-sdk-for-java/blob/main/sdk/search/azure-search-documents/CHANGELOG.md ↗

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in [Azure/azure-sdk-for-java/tree/main/sdk/search/azure-search-documents/src/samples](https://github.com/Azure/azure-sdk-for-java/tree/main/sdk/search/azure-search-documents/src/samples) ↗ on GitHub.

[] [Expand table](#)

Sample	Description
Index creation ↗	Create an index .
Indexer creation ↗	Create an indexer .
Data source creation ↗	Create a data source connection, which is required for indexer-based indexing of supported data sources .
Skillset creation ↗	Create a skillset that's attached to an indexer and perform AI-based enrichment during indexing.
Synonym creation ↗	Create a synonym map .

Sample	Description
Load documents	Upload or merge documents into an index in a data import operation.
Query syntax	Send a basic query .
Vector search	Create a vector field and send a vector query .

Doc samples

Code samples from the Azure AI Search team demonstrate features and workflows. The following samples are referenced in tutorials, quickstarts, and how-to articles that explain the code in detail. You can find these samples in [Azure-Samples/azure-search-java-samples](#) on GitHub.

 [Expand table](#)

Sample	Article	Description
quickstart	Quickstart: Full-text search	Create, load, and query an index using sample data.

Tip

Use the [samples browser](#) to search for Microsoft code samples on GitHub. You can filter your search by product, service, and language.

JavaScript samples for Azure AI Search

09/23/2025

Learn about JavaScript code samples that demonstrate the functionality and workflow of an Azure AI Search solution. These samples use the [Azure AI Search client library](#) for the [Azure SDK for JavaScript](#), which you can explore through the following links.

[+] [Expand table](#)

Target	Link
Package download	www.npmjs.com/package/@azure/search-documents ↗
API reference	@azure/search-documents
API test cases	github.com/Azure/azure-sdk-for-js/tree/main/sdk/search/search-documents/test ↗
Source code	github.com/Azure/azure-sdk-for-js/tree/main/sdk/search/search-documents ↗
Change log	github.com/Azure/azure-sdk-for-js/blob/main/sdk/search/search-documents/CHANGELOG.md ↗

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in [Azure/azure-sdk-for-js/tree/main/sdk/search/search-documents/samples](https://github.com/Azure/azure-sdk-for-js/tree/main/sdk/search/search-documents/samples) ↗ on GitHub.

JavaScript samples

[+] [Expand table](#)

Sample	Description
indexes ↗	Create, update, get, list, and delete indexes . This sample category also includes a service statistic sample.
indexers ↗	Create, update, get, list, reset, and delete indexers .
dataSourceConnections (for indexers) ↗	Create, update, get, list, and delete data source connections, which are required for indexer-based indexing of supported data sources .
skillsets ↗	Create, update, get, list, and delete skillsets that are attached to indexers and perform AI-based enrichment during indexing.

Sample	Description
synonymMaps	Create, update, get, list, and delete synonym maps .
vectorSearch	Index vectors and send a vector query .

TypeScript samples

[\[+\] Expand table](#)

Sample	Description
indexes	Create, update, get, list, and delete indexes . This sample category also includes a service statistic sample.
indexers	Create, update, get, list, reset, and delete indexers .
dataSourceConnections (for indexers)	Create, update, get, list, and delete data source connections, which are required for indexer-based indexing of supported data sources .
skillsets	Create, update, get, list, and delete skillsets that are attached to indexers and perform AI-based enrichment during indexing.
synonymMaps	Create, update, get, list, and delete synonym maps .
vectorSearch	Create, update, get, list, and delete vector search .

Doc samples

Code samples from the Azure AI Search team demonstrate features and workflows. The following samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in [Azure-Samples/azure-search-javascript-samples](#) on GitHub.

JavaScript samples

[\[+\] Expand table](#)

Sample	Article	Description
quickstart	Quickstart: Full-text search	Create, load, and query a search index using sample data.
quickstart-rag-js	Quickstart: Generative search (RAG)	Use grounding data from Azure AI Search with a chat completion model from Azure OpenAI.

Sample	Article	Description
quickstart-semantic-ranking-js	Quickstart: Semantic ranking	Add semantic ranking to an index schema and run semantic queries.
quickstart-vector-js	Quickstart: Vector search	Index and query vector content.

TypeScript samples

[] [Expand table](#)

Sample	Article	Description
quickstart-rag-ts	Quickstart: Generative search (RAG)	Use grounding data from Azure AI Search with a chat completion model from Azure OpenAI.
quickstart-semantic-ranking-ts	Quickstart: Semantic ranking	Add semantic ranking to an index schema and run semantic queries.
quickstart-vector-ts	Quickstart: Vector search	Index and query vector content.

Other samples

The following samples are also published by the Azure AI Search team but aren't referenced in documentation. Associated README files provide usage instructions.

[] [Expand table](#)

Sample	Description
azure-search-vector-sample.js	JavaScript example of how to perform vector search.
azure-function-search	JavaScript example of an Azure function that sends queries to a search service. You can substitute this JavaScript version for the <code>api</code> code used in Add search to web sites with .NET .
bulk-insert	JavaScript example of how to use the push APIs to upload and index documents.

Tip

Use the [samples browser](#) to search for Microsoft code samples on GitHub. You can filter your search by product, service, and language.

Python samples for Azure AI Search

Learn about Python code samples that demonstrate the functionality and workflow of an Azure AI Search solution. These samples use the [Azure AI Search client library](#) for the [Azure SDK for Python](#), which you can explore through the following links.

[+] [Expand table](#)

Target	Link
Package download	pypi.org/project/azure-search-documents/
API reference	azure-search-documents
API test cases	github.com/Azure/azure-sdk-for-python/tree/main/sdk/search/azure-search-documents/tests
Source code	github.com/Azure/azure-sdk-for-python/tree/main/sdk/search/azure-search-documents
Change log	github.com/Azure/azure-sdk-for-python/blob/main/sdk/search/azure-search-documents/CHANGELOG.md

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in [Azure/azure-sdk-for-python/tree/main/sdk/search/azure-search-documents/samples](https://github.com/Azure/azure-sdk-for-python/tree/main/sdk/search/azure-search-documents/samples) on GitHub.

Doc samples

Code samples from the Azure AI Search team demonstrate features and workflows. The following samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in [Azure-Samples/azure-search-python-samples](https://github.com/Azure-Samples/azure-search-python-samples) on GitHub.

[+] [Expand table](#)

Sample	Article	Description
Quickstart	Quickstart: Full-text search	Create, load, and query a search index using sample data.
Quickstart-Agentic-Retrieval	Quickstart: Agentic retrieval	Integrate semantic ranking with LLM-powered query planning and answer generation.

Sample	Article	Description
Quickstart-RAG ↗	Quickstart: Generative search (RAG)	Use grounding data from Azure AI Search with a chat completion model from Azure OpenAI.
Quickstart-Semantic-Search ↗	Quickstart: Semantic ranking	Add semantic ranking to an index schema and run semantic queries.
Quickstart-Vector-Search ↗	Quickstart: Vector search	Index and query vector content.
Tutorial-RAG ↗	Build a RAG solution using Azure AI Search	Create an indexing pipeline that loads, chunks, embeds, and ingests searchable content for RAG.
agentic-retrieval-pipeline-example ↗	Tutorial: Build an end-to-end agentic retrieval solution	Unlike Quickstart-Agentic-Retrieval ↗ , this sample incorporates Foundry Agent Service for request orchestration.

Accelerators

An accelerator is an end-to-end solution that includes code and documentation you can adapt for your own implementation of a specific scenario.

 [Expand table](#)

Sample	Description
rag-experiment-accelerator ↗	Conduct experiments and evaluations using Azure AI Search and the RAG pattern. This sample has code for loading multiple data sources, using various models, and creating various search indexes and queries.

Demos

A demo repo provides proof-of-concept source code for examples or scenarios shown in demonstrations. Unlike accelerators, demo solutions aren't designed for adaptation.

 [Expand table](#)

Sample	Description
azure-search-vector-samples ↗	Comprehensive collection of samples for vector search scenarios, organized by scenario or technology.

Sample	Description
azure-search-openai-demo	ChatGPT-like experience over enterprise data with Azure OpenAI Python code showing how to use Azure AI Search with large language models in Azure OpenAI. For background, see this blog post .
aisearch-openai-rag-audio	"Voice to RAG." This sample demonstrates a simple architecture for voice-based generative AI applications that enables Azure AI Search RAG on top of the real-time audio API with full-duplex audio streaming from client devices. It also securely handles access to both the model and the retrieval system. Backend code is written in Python, while frontend code is written in JavaScript. For an introduction, watch this video .

Other samples

The following samples are also published by the Azure AI Search team but aren't referenced in documentation. Associated README files provide usage instructions.

[Expand table](#)

Sample	Description
Quickstart-Document-Permissions-Pull-API	Using an indexer "pull API" approach, flow access control lists from a data source to search results and apply permission filters that restrict access to authorized content.
Quickstart-Document-Permissions-Push-API	Using the push APIs for indexing a JSON payload, flow embedded permission metadata to indexed documents and search results that are filtered based on user access to authorized content.
azure-function-search	Use an Azure function to send queries to a search service. You can substitute this Python version for the <code>api</code> code used in Add search to web sites with .NET .
bulk-insert	Use the push APIs to upload and index documents.
index-backup-and-restore.ipynb	Make a local copy of retrievable fields in an index and push those fields to a new index.
resumable-index-backup-restore	Back up and restore larger indexes that exceed 100,000 documents.



Use the [samples browser](#) to search for Microsoft code samples on GitHub. You can filter your search by product, service, and language.

Last updated on 11/21/2025

REST samples for Azure AI Search

Learn about REST API samples that demonstrate the functionality and workflow of an Azure AI Search solution. These samples use the [Search Service REST APIs](#).

REST is the definitive programming interface for Azure AI Search. All operations that can be invoked programmatically are first available in REST, followed by the SDKs. For this reason, most examples in our documentation use the REST APIs to demonstrate and explain important concepts.

You can use any client that supports HTTP calls. To learn how to formulate the HTTP request using Visual Studio Code with the REST Client extension, see the REST portion of [Quickstart: Full-text search](#).

Doc samples

Code samples from the Azure AI Search team demonstrate features and workflows. The following samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in [Azure-Samples/azure-search-rest-samples](#) ↗ on GitHub.

[] [Expand table](#)

Sample	Article	Description
quickstart ↗	Quickstart: Full-text search	Create, load, and query a search index using sample data.
quickstart-ACL ↗	Query-time ACL and RBAC enforcement	Implement query-time access control using role-based access control (RBAC) and access control lists (ACLs).
quickstart-agic-retrieval ↗	Quickstart: Agentic retrieval	Integrate semantic ranking with LLM-powered query planning and answer generation.
quickstart-RAG ↗	Quickstart: Classic generative search (RAG)	Use grounding data from Azure AI Search with a chat completion model from Azure OpenAI.
quickstart-semantic-search ↗	Quickstart: Semantic ranking	Add semantic ranking to an index schema and run semantic queries.
quickstart-vectors ↗	Quickstart: Vector search	Index and query vector content.
custom-analyzers ↗	Tutorial: Create a custom analyzer for phone numbers	Use an analyzer to preserve patterns and special characters in searchable content.
debug-sessions ↗	Tutorial: Fix a skillset using Debug Sessions	Create search objects that you later debug in the Azure portal.

Sample	Article	Description
index-json-blobs	Tutorial: Index JSON blobs from Azure Storage	Create an indexer, data source, and index for nested JSON within a JSON array. Demonstrates the jsonArray parsing model and documentRoot parameters.
knowledge-store	Create a knowledge store using REST	Populate a knowledge store for knowledge mining workflows.
projections	Define projections in a knowledge store	Specify the physical data structures in a knowledge store.
skillset-tutorial	Tutorial: AI-generated searchable content from Azure blobs	Create a skillset that iterates over Azure blobs to extract information and infer structure.

Other samples

Currently, there are no other REST samples available.



Use the [samples browser](#) to search for Microsoft code samples on GitHub. You can filter your search by product, service, and language.

Last updated on 11/21/2025

Transparency note: Azure AI Search

Important

Non-English translations are provided for convenience only. Please consult the [EN-US](#) version of this document for the definitive version.

What is a Transparency Note?

An AI system includes not only the technology, but also the people who will use it, the people who will be affected by it, and the environment in which it is deployed. Creating a system that is fit for its intended purpose requires an understanding of how technology works, what its capabilities and limitations are, and how to achieve the best performance. Microsoft's Transparency Notes are intended to help you understand how our AI technology works, the choices system owners can make that influence system performance and behavior, and the importance of thinking about the whole system, including the technology, the people, and the environment. You can use Transparency Notes when developing or deploying your own system or share them with the people who will use or be affected by your system.

Microsoft's Transparency Notes are part of a broader effort at Microsoft to put our AI Principles into practice. To find out more, see the [Microsoft AI principles](#).

The basics of Azure AI Search

Introduction

Azure AI Search gives developers tools, APIs, and SDKs for building a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications. Search is foundational to any application that surfaces data to users. Common scenarios include catalog or document search, online retail stores, or data exploration over proprietary content.

Searchable data can be in the form of text or vectors and ingested as-is from a data source or enriched by using AI to improve the overall search experience. Developers can convert data into numerical representations (called vectors), by choosing to call an external machine learning models (known as embedding models). Indexers can optionally include skill sets that support a powerful suite of data enrichment via several [Azure Language in Foundry Tools](#) capabilities, such as [Named Entity Recognition \(NER\)](#) and [personally identifiable information \(PII\) detection](#), and [Azure Vision in Foundry Tools](#) capabilities, including [optical character recognition \(OCR\)](#) and image analysis.

See the following tabs for more information about how Azure AI Search improves the search experience by using Foundry Tools or other AI systems to better understand the intent, semantics, and implied structure of a customer's content.

AI enrichment

[AI enrichment](#) is the application of machine learning models from Foundry Tools over content that is not easily searchable in its raw form. Through enrichment, analysis and inference are used to create searchable content and structure where none previously existed.

AI enrichment is an optional extension of the Azure AI Search indexer pipeline that connects to Foundry Tools in the same region as a customer's search service. An enrichment pipeline has the same core components as a typical indexer (indexer, data source, index), plus a skill set that specifies the atomic enrichment steps. A skill set can be assembled by using built-in skills based on the Foundry Tools APIs, such as [Vision](#) and [Language](#), or [custom skills](#) that run external code that you provide.

Capabilities

AI enrichment

System behavior

Several [built-in skills](#) for AI enrichment in Azure AI Search take advantage of Foundry Tools. See the Transparency Notes for each built-in skill linked below for considerations when choosing to use a skill:

- Key Phrase Extraction Skill: [Language - Key Phrase Extraction](#)
- Language Detection Skill: [Language - Language Detection](#)
- Entity Linking Skill: [Language - Entity Linking](#)
- Entity Recognition Skill: [Language - Named Entity Recognition \(NER\)](#)
- PII Detection Skill: [Language - PII Detection](#)
- Sentiment Skill: [Language - Sentiment Analysis](#)
- Image Analysis Skill: [Vision - Image Analysis](#)
- OCR Skill: [Vision - OCR](#)
- Document Layout Skill: [Document Intelligence](#)

See the documentation for each skill to learn more about their respective capabilities, limitations, performance, evaluations, and methods for integration and responsible use. Note that using these skills in combination may lead to compounding effects (for example, errors introduced when using OCR will carry through when using key phrase extraction).

Use cases

Example use cases

Because Azure AI Search is a full text search solution, the purpose of AI enrichment is to improve the search utility of unstructured content. Here are some examples of content enrichment scenarios supported by the built-in skills:

- **Translation** and **language detection** enable multilingual search.
- **Entity recognition** extracts **people**, **places**, and **other entities** from large chunks of text.
- **Key phrase extraction** identifies and then outputs important terms.
- **OCR** recognizes printed and handwritten text in binary files.
- **Image analysis** describes image content and outputs the descriptions as searchable text fields.
- **Integrated vectorization** is a preview feature that calls the Azure OpenAI embeddings model to vectorize data and store embeddings in Azure AI Search for similarity search.

Limitations

AI enrichment

AI enrichment in Azure AI Search uses the indexer and data source features of the service to call Foundry Tools to perform the content enrichment. Limitations of the indexers and data sources used in this process will apply. Review the [indexer and data source documentation](#) for more information about these related limitations. The limitations of each Foundry Tool used by the AI enrichment pipeline in Azure AI Search will also apply. See the [transparency notes for each service](#) for more information about these limitations.

Learn more about responsible AI

- [Microsoft AI principles ↗](#)

- Microsoft responsible AI resources ↗
- Microsoft Azure Learning courses on responsible AI

Learn more about Azure AI Search

- [Introduction to Azure AI Search](#)
 - [Feature descriptions](#)
 - [AI enrichment concepts](#)
 - [Retrieval Augmented Generation \(RAG\) in Azure AI Search](#)
-

Last updated on 08/22/2025

Azure AI Search REST API reference

09/24/2025

Azure AI Search (formerly known as *Azure Cognitive Search*) is a fully managed cloud search service that provides information retrieval over user-owned content.

Data plane REST APIs are used for indexing and query workflows, and are documented in this section.

Control plane operations for service administration are covered in a separate [Search Management REST API](#).

Versioned API docs

A version selector appears above the table of contents when you select an API reference article. The selector becomes available when you choose a page from the **Reference > Data Plane** folder.

The screenshot shows the Azure AI Search REST API reference page. On the left, there is a sidebar with a 'Version' dropdown menu containing the following options:

- 2025-09-01
- 2025-09-01
- 2025-08-01-preview
- 2025-05-01-preview

Below the dropdown is a tree view of API categories:

- Search Service
 - Overview
 - API versions
 - Data types
 - Data type map for indexers
 - Naming rules
 - HTTP status codes
 - HTTP request and response headers
 - OData support
- > Earlier versions
 - Reference
 - Data Plane
 - Overview
 - Data Sources

The main content area is titled 'Data Plane' and shows the following details:

- Learn / Azure / Azure REST API /
- Ask Learn
- Focus mode
- 9/24/2025
- Data Plane
- REST Operation Groups
- Expand table
- Operation Group
 - Data Sources
 - Documents
 - Get Service Statistics
 - Indexes
 - Indexes
 - Skillsets
 - Synonym Maps

Key concepts

Azure AI Search has the concepts of *search services*, *indexes*, *documents*, *indexers*, *data sources*, *skillsets*, and *synonym maps*.

- A search service hosts index, indexers, data sources, skillsets, and synonym maps as top-level objects.

- A search index provides persistent storage of search documents. Search documents are your data, articulated as a collection of fields, loaded from external sources and pushed to an index to make it searchable.
- A search indexer adds automation, reading data in native formats and serializing it into JSON.
- An indexer has a data source and points to an index.
- An indexer might also have a skillset that adds [AI enrichment](#) and [integrated vectorization](#) to the indexing pipeline. Skillsets are always attached to an indexer. They invoke machine learning to extract or chunk text, vectorize content, infer features, or add structure to content for improved indexing by a search service.

Altogether, you can create the following objects on a search service:

[] [Expand table](#)

Objects	Description
Data sources	A data source connection used by an indexer to retrieve and refresh documents for indexing. Data sources have a <code>type</code> . You can use the Microsoft-provided connections for Azure, or partner connectors. See Data sources gallery for the full list.
Documents	Conceptually, a document is an entity in your index. Mapping this concept to more familiar database equivalents: a search index equates to a table, and documents are roughly equivalent to rows in a table. Documents exist only in an index, and are retrieved only through queries that target the documents collection (<code>/docs</code>) of an index. All operations performed on the collection such as uploading, merging, deleting, or querying documents take place in the context of a single index, so the URL format document operations always includes <code>/indexes/[index name]/docs</code> for a given index name.
Indexes	An index is stored on your search service and populated with JSON documents that are indexed and tokenized for information retrieval. The fields collection of an index defines the structure of the search document. Fields have a name, data types, and attributes that determine usage. For example, <code>searchable</code> fields are used in full text search, and thus tokenized during indexing. An index also defines other constructs, such as scoring profiles for relevance tuning, suggesters, semantic configurations, and custom analyzers.
Indexers	Indexers provide indexing automation. An indexer connects to a data source, reads in the data, and passes it to a search engine for indexing into a target search index. Indexers read from an external source using connection information in a data source, and serialize the incoming data into JSON search documents. In addition to a data source, an indexer also requires an index. The index specifies the fields and attributes of the search documents.
Skillsets	A skillset adds external processing steps to indexer execution, and is used for applying AI or deep learning models to analyze or transform content for improved searchability in an index. The contents of a skillset are one or more <i>skills</i> , which can be built-in skills created by Microsoft, custom skills, or a combination of both. Built-in skills exist for image analysis, including OCR, and natural language processing. Other examples of built-in skills include entity recognition, key phrase extraction, chunking text into logical pages, among others. A

Objects	Description
	skillset is high-level standalone object that exists on a level equivalent to indexes, indexers, and data sources, but it's operational only within indexer processing. As a high-level object, you can design a skillset once, and then reference it in multiple indexers.
Synonym maps	A synonym map is service-level object that contains user-defined synonyms. This object is maintained independently from search indexes. Once uploaded, you can point any searchable field to the synonym map (one per field).

Permissions and access control

You can use key-based authentication or role-based through Microsoft Entra ID.

- [Key-based authentication](#) relies on API keys that are generated for the search service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it. You can use an [Admin API key](#) for read-write operations or a [Query API key](#) for read access to the documents collection of a search index.
- [Microsoft Entra ID authentication and role-based access control](#) requires that you have an established tenant in Microsoft Entra ID, with security principals and role assignments. Members of the following roles have data plane access. You can create custom roles if the built-in roles are insufficient.

When you use roles on the connection, your client app presents a bearer token in the authorization header. See [Authorize access to a search app using Microsoft Entra ID](#) for help with setting this up.

You can disable key-based authentication or role-based authentication. If you disable role-based authentication, it only applies to data plane operations. Control plane operations, such as service administration, always use role-based authentication. See [Microsoft Entra ID authentication and role-based access control for Azure AI Search](#) for details.

Calling the APIs

The APIs documented in this section provide access to operations on search data, such as index creation and population, document upload, and queries. When calling APIs, keep the following points in mind:

- Requests must be issued over HTTPS (on the default port 443).
- Request URIs must include the **api-version**. The value must be set to a supported version, formatted as shown in this example: `GET https://[search service]`

```
name].search.windows.net/indexes?api-version=2024-07-01
```

- Request headers must include either an api-key or a bearer token for authenticated connections. Optionally, you can set the Accept HTTP header. If the content type header isn't set, the default is assumed to be application/json.

See also

- [Quickstart: Vector search using REST APIs](#)
- [Quickstart: Keyword search using REST APIs](#)
- [Create a search service in Azure portal](#)
- [Common HTTP request and response headers](#)

Management REST API reference (Azure AI Search)

08/01/2025

The Management REST APIs of **Azure AI Search** (formerly known as *Azure Cognitive Search*) provide programmatic access to administrative operations:

- Create, update, and delete a search service
- Retrieve search service information
- Create, regenerate, or retrieve `api-keys` (query or admin keys)
- Add or remove replicas and partitions (adjust capacity)
- Configure a search service to use a private endpoint
- Enable or disable Azure role-based access control

For data plane tasks, such as creating and querying an index, use the [Search Service REST APIs](#) instead.

How to work with the management REST APIs

To fully administer your service programmatically, you work with two APIs: the Management REST API of Azure AI Search documented here, plus the common [Azure Resource Manager REST API](#).

The Resource Manager API is used for general-purpose operations that aren't service specific, such as querying subscription data, listing geo-locations, and so forth. All Resource Manager API calls are authenticated using Microsoft Entra ID. You must have membership and a role assignment in a Microsoft Entra tenant to make Resource Manager API calls.

Versioned API docs

REST API docs are versioned. When you open an API reference page, select the API version from the dropdown filter.

The screenshot shows the Azure API Management interface. On the left, there's a sidebar with a 'Version' dropdown and a tree view of service management categories. The 'Admin Keys' node under 'Resource Manager' is selected and highlighted with a red box. A main content area on the right displays the 'Admin Keys - Get' operation details, including the service name ('Search Management'), API version ('2025-05-01'), and a brief description ('Gets the primary and secondary admin API keys for the specified Azure AI Search service'). Below this is an 'HTTP' section with a POST request template: `POST https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Search/searchServices/{serviceName}?api-version=2020-08-01`. There are 'Copy' and 'Try It' buttons next to the template. At the bottom, there's a table for 'URI Parameters'.

Name	In	Required	Type	Description
resourceGroupName	path	True	string	The name of the resource group within the current subscription. You can obtain this value from the Azure Resource Manager API or the portal.

Permissions and access control

Administration rights are conveyed through built-in roles in Microsoft Entra ID:

- **Owner or Contributor** assignments are required for most tasks.
- **Reader** role has limited access to service information (GET operations, except for API keys, which require Owner or Contributor permissions).

For more information, see [Role-based access control in Azure AI Search](#).

A best practice is to assign roles to groups rather than individual users. An Azure subscription owner or global administrator manages these assignments. For more information, see [Assign Microsoft Entra roles to groups](#).

Connect to the management endpoint

When setting up the connection, start with the Resource Manager endpoint

`https://management.azure.com` and then add the subscription ID, provider (`Microsoft.Search`), and API version. Operations are specified in the body of the request.

A fully specified endpoint has the following components:

The screenshot shows the 'HTTP' section of the API operation details. It displays the full URL template: `https://management.azure.com/subscriptions/[subscriptionId]/resourceGroups/[resourceGroupName]/providers/Microsoft.Search/searchServices/[serviceName]?api-version=2020-08-01`.

The following clients are commonly used to call the management REST APIs:

Client	Instructions and examples
Azure PowerShell using the Az.Search module	Manage Azure AI Search using Azure PowerShell
Azure Command-Line Interface (CLI)	Manage Azure AI Search using the Azure CLI
Visual Studio Code  with the REST client 	Manage Azure AI Search using REST

You can also [manage a search service using the Azure portal](#).

Management APIs in other SDKs

The REST APIs are foundational to Azure AI Search, and any changes to the programming model are reflected in the REST APIs first.

Alternatives include the following Azure SDKs, which are independently updated on separate release schedules:

- [Microsoft Azure AI Search management client library for .NET](#)
- [Microsoft Azure SDK for Python](#)
- [Azure Search Management client library for JavaScript](#)
- [Azure Resource Manager Azure AI Search client library for Java](#)

See also

- [Azure AI Search documentation](#)
- [Manage Azure AI Search using REST](#)
- [Manage Azure AI Search using Azure PowerShell](#)
- [Manage Azure AI Search using Azure CLI](#)
- [Manage Azure AI Search using Azure portal](#)

Azure AI Search client library for .NET - version 11.7.0

[Azure AI Search](#) (formerly known as "Azure Cognitive Search") is an AI-powered information retrieval platform that helps developers build rich search experiences and generative AI apps that combine large language models with enterprise data.

The Azure AI Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and unstructured documents. A skillset leverages APIs from Azure AI Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines and chat-style apps.

Use the `Azure.Search.Documents` client library to:

- Submit queries using vector, keyword, and hybrid query forms.
- Implement filtered queries for metadata, geospatial search, faceted navigation, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through semantic ranking and scoring profiles to factor in business logic or freshness.

[Source code](#) | [Package \(NuGet\)](#) | [API reference documentation](#) | [REST API documentation](#) | [Product documentation](#) | [Samples](#)

Getting started

Install the package

Install the Azure AI Search client library for .NET with [NuGet](#):

.NET CLI

```
dotnet add package Azure.Search.Documents
```

Prerequisites

You need an [Azure subscription](#) and a [search service](#) to use this package.

To create a new search service, you can use the [Azure portal](#), [Azure PowerShell](#), or the [Azure CLI](#). Here's an example using the Azure CLI to create a free instance for getting started:

Powershell

```
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free  
--location westus
```

See [choosing a pricing tier](#) for more information about available options.

Authenticate the client

To interact with the search service, you'll need to create an instance of the appropriate client class: `SearchClient` for searching indexed documents, `SearchIndexClient` for managing indexes, or `SearchIndexerClient` for crawling data sources and loading search documents into an index. To instantiate a client object, you'll need an [endpoint](#) and [Azure roles](#) or an [API key](#). You can refer to the documentation for more information on [supported authenticating approaches](#) with the search service.

Get an API Key

An API key can be an easier approach to start with because it doesn't require pre-existing role assignments.

You can get the [endpoint](#) and an [API key](#) from the search service in the [Azure portal](#). Please refer the [documentation](#) for instructions on how to get an API key.

Alternatively, you can use the following [Azure CLI](#) command to retrieve the API key from the search service:

Powershell

```
az search admin-key show --service-name <mysearch> --resource-group <mysearch-rg>
```

There are two types of keys used to access your search service: **admin** (*read-write*) and **query** (*read-only*) keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

Create a SearchClient

To instantiate the `SearchClient`, you'll need the **endpoint**, **API key** and **index name**:

C#

```
string indexName = "nycjobs";

// Get the service endpoint and API key from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");

// Create a client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchClient client = new SearchClient(endpoint, indexName, credential);
```

Create a client using Microsoft Entra ID authentication

You can also create a `SearchClient`, `SearchIndexClient`, or `SearchIndexerClient` using Microsoft Entra ID authentication. Your user or service principal must be assigned the "Search Index Data Reader" role. Using the [DefaultAzureCredential](#) you can authenticate a service using Managed Identity or a service principal, authenticate as a developer working on an application, and more all without changing code. Please refer the [documentation](#) for instructions on how to connect to Azure AI Search using Azure role-based access control (Azure RBAC).

Before you can use the `DefaultAzureCredential`, or any credential type from [Azure.Identity](#), you'll first need to [install the Azure.Identity package](#).

To use `DefaultAzureCredential` with a client ID and secret, you'll need to set the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables; alternatively, you can pass those values to the `ClientSecretCredential` also in `Azure.Identity`.

Make sure you use the right namespace for `DefaultAzureCredential` at the top of your source file:

C#

```
using Azure.Identity;
```

Then you can create an instance of `DefaultAzureCredential` and pass it to a new instance of your client:

C#

```
string indexName = "nycjobs";

// Get the service endpoint from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
DefaultAzureCredential credential = new DefaultAzureCredential();

// Create a client
SearchClient client = new SearchClient(endpoint, indexName, credential);
```

ASP.NET Core

To inject `SearchClient` as a dependency in an ASP.NET Core app, first install the package `Microsoft.Extensions.Azure`. Then register the client in the `Startup.ConfigureServices` method:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAzureClients(builder =>
    {
        builder.AddSearchClient(Configuration.GetSection("SearchClient"));
    });

    services.AddControllers();
}
```

To use the preceding code, add this to your configuration:

JSON

```
{
  "SearchClient": {
    "endpoint": "https://<resource-name>.search.windows.net",
    "indexname": "nycjobs"
  }
}
```

You'll also need to provide your resource key to authenticate the client, but you shouldn't be putting that information in the configuration. Instead, when in development, use [User-Secrets](#). Add the following to `secrets.json`:

JSON

```
{  
  "SearchClient": {  
    "credential": { "key": "<you resource key>" }  
  }  
}
```

When running in production, it's preferable to use [environment variables](#):

```
SEARCH__CREDENTIAL__KEY="..."
```

Or use other secure ways of storing secrets like [Azure Key Vault](#).

For more details about Dependency Injection in ASP.NET Core apps, see [Dependency injection with the Azure SDK for .NET](#).

Key concepts

An Azure AI Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. (*If you're brand new to search, you can make a very rough analogy between indexes and database tables.*) The `Azure.Search.Documents` client library exposes operations on these resources through three main client types.

- `SearchClient` helps with:
 - [Searching](#) your indexed documents using [vector queries](#), [keyword queries](#) and [hybrid queries](#)
 - [Vector query filters](#) and [Text query filters](#)
 - [Semantic ranking](#) and [scoring profiles](#) for boosting relevance
 - [Autocompleting](#) partially typed search terms based on documents in the index
 - [Suggesting](#) the most likely matching text in documents as a user types
 - [Adding, Updating or Deleting Documents](#) documents from an index
- `SearchIndexClient` allows you to:
 - [Create, delete, update, or configure a search index](#)
 - [Declare custom synonym maps to expand or rewrite queries](#)
- `SearchIndexerClient` allows you to:

- Create indexers to automatically crawl data sources
- Define AI powered Skillsets to transform and enrich your data

Azure AI Search provides two powerful features:

Semantic ranking

Semantic ranking enhances the quality of search results for text-based queries. By enabling semantic ranking on your search service, you can improve the relevance of search results in two ways:

- It applies secondary ranking to the initial result set, promoting the most semantically relevant results to the top.
- It extracts and returns captions and answers in the response, which can be displayed on a search page to enhance the user's search experience.

To learn more about semantic ranking, you can refer to the [sample ↗](#).

Additionally, for more comprehensive information about semantic ranking, including its concepts and usage, you can refer to the [documentation](#). The documentation provides in-depth explanations and guidance on leveraging the power of semantic ranking in Azure AI Search.

Vector search

Vector search is an information retrieval technique that uses numeric representations of searchable documents and query strings. By searching for numeric representations of content that are most similar to the numeric query, vector search can find relevant matches, even if the exact terms of the query are not present in the index. Moreover, vector search can be applied to various types of content, including images and videos and translated text, not just same-language text.

To learn how to index vector fields and perform vector search, you can refer to the [sample ↗](#). This sample provides detailed guidance on indexing vector fields and demonstrates how to perform vector search.

Additionally, for more comprehensive information about vector search, including its concepts and usage, you can refer to the [documentation](#). The documentation provides in-depth explanations and guidance on leveraging the power of vector search in Azure AI Search.

The `Azure.Search.Documents` client library (v11) provides APIs for data plane operations. The previous `Microsoft.Azure.Search` client library (v10) is now retired. It has many similar looking APIs, so please be careful to avoid confusion when exploring online resources. A good rule of

thumb is to check for the namespace `using Azure.Search.Documents`; when you're looking for API reference.

Thread safety

We guarantee that all client instance methods are thread-safe and independent of each other ([guideline ↗](#)). This ensures that the recommendation of reusing client instances is always safe, even across threads.

Additional concepts

[Client options ↗](#) | [Accessing the response ↗](#) | [Long-running operations ↗](#) | [Handling failures ↗](#) | [Diagnostics ↗](#) | [Mocking](#) | [Client lifetime ↗](#)

Examples

The following examples all use a simple [Hotel data set ↗](#) that you can [import into your own index from the Azure portal](#). These are just a few of the basics - please [check out our Samples ↗](#) for much more.

- [Querying](#)
 - [Use C# types for search results](#)
 - [Use SearchDocument like a dictionary for search results](#)
 - [SearchOptions](#)
- [Creating an index](#)
- [Adding documents to your index](#)
- [Retrieving a specific document from your index](#)
- [Async APIs](#)

Advanced authentication

- [Create a client that can authenticate in a national cloud](#)

Querying

Let's start by importing our namespaces.

C#

```
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
```

```
using Azure.Core.GeoJson;
```

We'll then create a `SearchClient` to access our hotels search index.

C#

```
// Get the service endpoint and API key from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");
string indexName = "hotels";

// Create a client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchClient client = new SearchClient(endpoint, indexName, credential);
```

There are two ways to interact with the data returned from a search query. Let's explore them with a search for a "luxury" hotel.

Use C# types for search results

We can decorate our own C# types with [attributes from System.Text.Json](#):

C#

```
public class Hotel
{
    [JsonPropertyName("HotelId")]
    [SimpleField(IsKey = true, IsFilterable = true, IsSortable = true)]
    public string Id { get; set; }

    [JsonPropertyName("HotelName")]
    [SearchableField(IsFilterable = true, IsSortable = true)]
    public string Name { get; set; }

    [SimpleField(IsFilterable = true, IsSortable = true)]
    public GeoPoint GeoLocation { get; set; }

    // Complex fields are included automatically in an index if not ignored.
    public HotelAddress Address { get; set; }
}

public class HotelAddress
{
    public string StreetAddress { get; set; }

    [SimpleField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
    public string City { get; set; }

    [SimpleField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
    public string StateProvince { get; set; }
```

```
[SimpleField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
public string Country { get; set; }

[SimpleField(IsFilterable = true, IsSortable = true, IsFacetable = true)]
public string PostalCode { get; set; }
}
```

Then we use them as the type parameter when querying to return strongly-typed search results:

C#

```
SearchResults<Hotel> response = client.Search<Hotel>("luxury");
foreach (SearchResult<Hotel> result in response.GetResults())
{
    Hotel doc = result.Document;
    Console.WriteLine($"{doc.Id}: {doc.Name}");
}
```

If you're working with a search index and know the schema, creating C# types is recommended.

Use `SearchDocument` like a dictionary for search results

If you don't have your own type for search results, `SearchDocument` can be used instead. Here we perform the search, enumerate over the results, and extract data using `SearchDocument`'s dictionary indexer.

C#

```
SearchResults<SearchDocument> response = client.Search<SearchDocument>("luxury");
foreach (SearchResult<SearchDocument> result in response.GetResults())
{
    SearchDocument doc = result.Document;
    string id = (string)doc["HotelId"];
    string name = (string)doc["HotelName"];
    Console.WriteLine($"{id}: {name}");
}
```

SearchOptions

The `SearchOptions` provide powerful control over the behavior of our queries. Let's search for the top 5 luxury hotels with a good rating.

C#

```
int stars = 4;
SearchOptions options = new SearchOptions
{
    // Filter to only Rating greater than or equal our preference
    Filter = SearchFilter.Create($"Rating ge {stars}"),
    Size = 5, // Take only 5 results
    OrderBy = { "Rating desc" } // Sort by Rating from high to low
};
SearchResults<Hotel> response = client.Search<Hotel>("luxury", options);
// ...
```

Creating an index

You can use the `SearchIndexClient` to create a search index. Fields can be defined from a model class using `FieldBuilder`. Indexes can also define suggesters, lexical analyzers, and more.

Using the [Hotel sample](#) above, which defines both simple and complex fields:

C#

```
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");

// Create a service client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchIndexClient client = new SearchIndexClient(endpoint, credential);

// Create the index using FieldBuilder.
SearchIndex index = new SearchIndex("hotels")
{
    Fields = new FieldBuilder().Build(typeof(Hotel)),
    Suggesters =
    {
        // Suggest query terms from the HotelName field.
        new SearchSuggester("sg", "HotelName")
    }
};

client.CreateIndex(index);
```

In scenarios when the model is not known or cannot be modified, you can also create fields explicitly using convenient `SimpleField`, `SearchableField`, or `ComplexField` classes:

C#

```
// Create the index using field definitions.
SearchIndex index = new SearchIndex("hotels")
{
```

```

Fields =
{
    new SimpleField("HotelId", SearchFieldDataType.String) { IsKey = true,
IsFilterable = true, IsSortable = true },
    new SearchableField("HotelName") { IsFilterable = true, IsSortable = true },
    new SearchableField("Description") { AnalyzerName =
LexicalAnalyzerName.EnLucene },
    new SearchableField("Tags", collection: true) { IsFilterable = true,
IsFacetable = true },
    new ComplexField("Address")
{
    Fields =
{
        new SearchableField("StreetAddress"),
        new SearchableField("City") { IsFilterable = true, IsSortable =
true, IsFacetable = true },
        new SearchableField("StateProvince") { IsFilterable = true,
IsSortable = true, IsFacetable = true },
        new SearchableField("Country") { IsFilterable = true, IsSortable =
true, IsFacetable = true },
        new SearchableField("PostalCode") { IsFilterable = true, IsSortable =
true, IsFacetable = true }
    }
},
Suggesters =
{
    // Suggest query terms from the hotelName field.
    new SearchSuggester("sg", "HotelName")
}
};

client.CreateIndex(index);

```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are [a few special rules for merging](#) to be aware of.

C#

```

IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
    IndexDocumentsAction.Upload(new Hotel { Id = "783", Name = "Upload Inn" }),
    IndexDocumentsAction.Merge(new Hotel { Id = "12", Name = "Renovated Ranch" }));

IndexDocumentsOptions options = new IndexDocumentsOptions { ThrowOnAnyError = true
};
client.IndexDocuments(batch, options);

```

The request will succeed even if any of the individual actions fail and return an `IndexDocumentsResult` for inspection. There's also a `ThrowOnAnyError` option if you only care

about success or failure of the whole batch.

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

```
C#
```

```
Hotel doc = client.GetDocument<Hotel>("1");
Console.WriteLine($"{doc.Id}: {doc.Name}");
```

Async APIs

All of the examples so far have been using synchronous APIs, but we provide full support for async APIs as well. You'll generally just add an `Async` suffix to the name of the method and `await` it.

```
C#
```

```
SearchResults<Hotel> searchResponse = await client.SearchAsync<Hotel>("luxury");
await foreach (SearchResult<Hotel> result in searchResponse.GetResultsAsync())
{
    Hotel doc = result.Document;
    Console.WriteLine($"{doc.Id}: {doc.Name}");
}
```

Authenticate in a National Cloud

To authenticate in a [National Cloud](#), you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `Audience` in `SearchClientOptions`

```
C#
```

```
// Create a SearchClient that will authenticate through AAD in the China national
cloud
string indexName = "nycjobs";
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
```

```
SearchClient client = new SearchClient(endpoint, indexName,
    new DefaultAzureCredential(
        new DefaultAzureCredentialOptions()
    {
        AuthorityHost = AzureAuthorityHosts.AzureChina
    }),
    new SearchClientOptions()
{
    Audience = SearchAudience.AzureChina
});
```

Troubleshooting

Any `Azure.Search.Documents` operation that fails will throw a [RequestFailedException](#) with helpful [Status codes](#). Many of these errors are recoverable.

C#

```
try
{
    return client.GetDocument<Hotel>("12345");
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    Console.WriteLine("We couldn't find the hotel you are looking for!");
    Console.WriteLine("Please try selecting another.");
    return null;
}
```

You can also easily [enable console logging](#) if you want to dig deeper into the requests you're making against the service.

See our [troubleshooting guide](#) for details on how to diagnose various failure scenarios.

Next steps

- Go further with `Azure.Search.Documents` and our [samples](#)
- Read more about the [Azure AI Search service](#)

Contributing

See our [Search CONTRIBUTING.md](#) for details on building, testing, and contributing to this library.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit cla.microsoft.com.

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information see the [Code of Conduct FAQ](#) or contact opencode@microsoft.com with any additional questions or comments.

Last updated on 10/09/2025

Microsoft Azure AI Search management client library for .NET

07/29/2025

Azure AI Search (formerly known as "Azure Cognitive Search") is an AI-powered information retrieval platform that helps developers build rich search experiences and generative AI apps that combine large language models with enterprise data.

This library supports managing Microsoft Azure AI Search resources. Use this library to create, configure, and manage your search resources.

This library follows the [new Azure SDK guidelines](#), and provides many core capabilities:

- Support MSAL.NET, Azure.Identity is out of box for supporting MSAL.NET.
- Support [OpenTelemetry](<https://opentelemetry.io/>) for distributed tracing.
- HTTP pipeline with custom policies.
- Better error-handling.
- Support uniform telemetry across all languages.

Getting started

Install the package

Install the Microsoft Azure AI Search management library for .NET with [NuGet](#):

.NET CLI

```
dotnet add package Azure.ResourceManager.Search
```

Prerequisites

- You must have an [Microsoft Azure subscription](#).

Authenticate the Client

To create an authenticated client and start interacting with Microsoft Azure resources, see the [quickstart guide here](#).

Key concepts

Key concepts of the Microsoft Azure SDK for .NET can be found [here](#).

Documentation

Documentation is available to help you learn how to use this package:

- [Quickstart](#).
- [API References](#).
- [Authentication](#).

Examples

Code samples for using the management library for .NET can be found in the following locations

- [.NET Management Library Code Samples](#)

Troubleshooting

- File an issue via [GitHub Issues](#).
- Check [previous questions](#) or ask new ones on Stack Overflow using Azure and .NET tags.

Next steps

For more information about Microsoft Azure SDK, see [this website](#).

Contributing

For details on contributing to this repository, see the [contributing guide](#).

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit <https://cla.microsoft.com>.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (for example, label, comment). Follow the

instructions provided by the bot. You'll only need to do this action once across all repositories using our CLA.

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information, see the [Code of Conduct FAQ](#) or contact opencode@microsoft.com with any other questions or comments.

Azure AI Search client library for Java - version 11.8.0

This is the Java client library for [Azure AI Search](#) (formerly known as "Azure Cognitive Search"). Azure AI Search service is an AI-powered information retrieval platform that helps developers build rich search experiences and generative AI apps that combine large language models with enterprise data.

Azure AI Search is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and unstructured documents. A skillset leverages APIs from Azure AI Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines and chat-style apps.

Use the Azure AI Search client library to:

- Submit queries using vector, keyword, and hybrid query forms.
- Implement filtered queries for metadata, geospatial search, faceted navigation, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through semantic ranking and scoring profiles to factor in business logic or freshness.

[Source code ↗](#) | [Package \(Maven\) ↗](#) | [API reference documentation ↗](#) | [Product documentation](#) | [Samples ↗](#)

Getting started

Include the package

Include the BOM file

Please include the azure-sdk-bom to your project to take dependency on the General Availability (GA) version of the library. In the following snippet, replace the {bom_version_to_target} placeholder with the version number. To learn more about the BOM, see the [AZURE SDK BOM README](#).

XML

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-sdk-bom</artifactId>
      <version>{bom_version_to_target}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

and then include the direct dependency in the dependencies section without the version tag.

XML

```
<dependencies>
  <dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-search-documents</artifactId>
  </dependency>
</dependencies>
```

Include direct dependency

If you want to take dependency on a particular version of the library that is not present in the BOM, add the direct dependency to your project as follows.

XML

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-search-documents</artifactId>
  <version>11.8.0</version>
</dependency>
```

Prerequisites

- Java Development Kit (JDK) with version 8 or above
 - Here are details about Java 8 client compatibility with Azure Certificate Authority.
- Azure subscription ↗
- Azure AI Search service ↗
- To create a new search service, you can use the [Azure portal](#), [Azure PowerShell](#), or the [Azure CLI](#). Here's an example using the Azure CLI to create a free instance for getting started:

Bash

```
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free  
--location westus
```

See [choosing a pricing tier](#) for more information about available options.

Authenticate the client

To interact with the Search service, you'll need to create an instance of the appropriate client class: `SearchClient` for searching indexed documents, `SearchIndexClient` for managing indexes, or `SearchIndexerClient` for crawling data sources and loading search documents into an index. To instantiate a client object, you'll need an [endpoint](#) and [Azure roles](#) or an [API key](#). You can refer to the documentation for more information on [supported authenticating approaches](#) with the Search service.

Get an API Key

An API key can be an easier approach to start with because it doesn't require pre-existing role assignments.

You can get the [endpoint](#) and an [API key](#) from the search service in the [Azure Portal](#) ↗. Please refer the [documentation](#) for instructions on how to get an API key.

Alternatively, you can use the following [Azure CLI](#) command to retrieve the API key from the search service:

Bash

```
az search admin-key show --service-name <mysearch> --resource-group <mysearch-rg>
```

Note:

- The example Azure CLI snippet above retrieves an admin key. This allows for easier access when exploring APIs, but it should be managed carefully.

- There are two types of keys used to access your search service: **admin** (*read-write*) and **query** (*read-only*) keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

The SDK provides three clients.

- `SearchIndexClient` for CRUD operations on indexes and synonym maps.
- `SearchIndexerClient` for CRUD operations on indexers, data sources, and skillsets.
- `SearchClient` for all document operations.

Create a SearchIndexClient

To create a `SearchIndexClient/SearchIndexAsyncClient`, you will need the values of the Azure AI Search service URL endpoint and admin key.

Java

```
SearchIndexClient searchIndexClient = new SearchIndexClientBuilder()  
    .endpoint(ENDPOINT)  
    .credential(new AzureKeyCredential(API_KEY))  
    .buildClient();
```

or

Java

```
SearchIndexAsyncClient searchIndexAsyncClient = new SearchIndexClientBuilder()  
    .endpoint(ENDPOINT)  
    .credential(new AzureKeyCredential(API_KEY))  
    .buildAsyncClient();
```

Create a SearchIndexerClient

To create a `SearchIndexerClient/SearchIndexerAsyncClient`, you will need the values of the Azure AI Search service URL endpoint and admin key.

Java

```
SearchIndexerClient searchIndexerClient = new SearchIndexerClientBuilder()  
    .endpoint(ENDPOINT)  
    .credential(new AzureKeyCredential(API_KEY))  
    .buildClient();
```

or

Java

```
SearchIndexerAsyncClient searchIndexerAsyncClient = new SearchIndexerClientBuilder()
    .endpoint(ENDPOINT)
    .credential(new AzureKeyCredential(API_KEY))
    .buildAsyncClient();
```

Create a SearchClient

Once you have the values of the Azure AI Search service URL endpoint and admin key, you can create the `SearchClient/SearchAsyncClient` with an existing index name:

Java

```
SearchClient searchClient = new SearchClientBuilder()
    .endpoint(ENDPOINT)
    .credential(new AzureKeyCredential(ADMIN_KEY))
    .indexName(INDEX_NAME)
    .buildClient();
```

or

Java

```
SearchAsyncClient searchAsyncClient = new SearchClientBuilder()
    .endpoint(ENDPOINT)
    .credential(new AzureKeyCredential(ADMIN_KEY))
    .indexName(INDEX_NAME)
    .buildAsyncClient();
```

Create a client using Microsoft Entra ID authentication

You can also create a `SearchClient`, `SearchIndexClient`, or `SearchIndexerClient` using Microsoft Entra ID authentication. Your user or service principal must be assigned the "Search Index Data Reader" role. Using the [DefaultAzureCredential](#) you can authenticate a service using Managed Identity or a service principal, authenticate as a developer working on an application, and more all without changing code. Please refer the [documentation](#) for instructions on how to connect to Azure AI Search using Azure role-based access control (Azure RBAC).

Before you can use the `DefaultAzureCredential`, or any credential type from [Azure.Identity](#), you'll first need to [install the Azure.Identity package](#).

To use `DefaultAzureCredential` with a client ID and secret, you'll need to set the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables;

alternatively, you can pass those values to the `ClientSecretCredential` also in `azure-identity`.

Make sure you use the right namespace for `DefaultAzureCredential` at the top of your source file:

Java

```
import com.azure.identity.DefaultAzureCredential;
import com.azure.identity.DefaultAzureCredentialBuilder;
```

Then you can create an instance of `DefaultAzureCredential` and pass it to a new instance of your client:

Java

```
String indexName = "nycjobs";

// Get the service endpoint from the environment
String endpoint = Configuration.getGlobalConfiguration().get("SEARCH_ENDPOINT");
DefaultAzureCredential credential = new DefaultAzureCredentialBuilder().build();

// Create a client
SearchClient client = new SearchClientBuilder()
    .endpoint(endpoint)
    .indexName(indexName)
    .credential(credential)
    .buildClient();
```

Send your first search query

To get running with Azure AI Search first create an index following this [guide](#). With an index created you can use the following samples to begin using the SDK.

Key concepts

An Azure AI Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. (*If you're new to search, you can make a very rough analogy between indexes and database tables.*) The `azure-search-documents` client library exposes operations on these resources through two main client types.

- `SearchClient` helps with:
 - [Searching](#) your indexed documents using [vector queries](#), [keyword queries](#) and [hybrid queries](#)
 - [Vector query filters](#) and [Text query filters](#)
 - [Semantic ranking](#) and [scoring profiles](#) for boosting relevance

- Autocompleting partially typed search terms based on documents in the index
- Suggesting the most likely matching text in documents as a user types
- Adding, Updating or Deleting Documents documents from an index
- `SearchIndexClient` allows you to:
 - Create, delete, update, or configure a search index
 - Declare custom synonym maps to expand or rewrite queries
- `SearchIndexerClient` allows you to:
 - Start indexers to automatically crawl data sources
 - Define AI powered Skillsets to transform and enrich your data

Azure AI Search provides two powerful features:

Semantic ranking

Semantic ranking enhances the quality of search results for text-based queries. By enabling semantic ranking on your search service, you can improve the relevance of search results in two ways:

- It applies secondary ranking to the initial result set, promoting the most semantically relevant results to the top.
- It extracts and returns captions and answers in the response, which can be displayed on a search page to enhance the user's search experience.

To learn more about semantic ranking, you can refer to the [documentation](#).

Vector Search

Vector search is an information retrieval technique that uses numeric representations of searchable documents and query strings. By searching for numeric representations of content that are most similar to the numeric query, vector search can find relevant matches, even if the exact terms of the query are not present in the index. Moreover, vector search can be applied to various types of content, including images and videos and translated text, not just same-language text.

To learn how to index vector fields and perform vector search, you can refer to the [sample ↗](#). This sample provides detailed guidance on indexing vector fields and demonstrates how to perform vector search.

Additionally, for more comprehensive information about vector search, including its concepts and usage, you can refer to the [documentation](#). The documentation provides in-depth explanations and guidance on leveraging the power of vector search in Azure AI Search.

Examples

The following examples all use a simple [Hotel data set](#) that you can import into your own index from the [Azure portal](#). These are just a few of the basics - please check out our [Samples](#) for much more.

- [Querying](#)
 - Use `SearchDocument` like a dictionary for search results
 - Use Java model for search results
 - Search Options
- [Creating an index](#)
- [Adding documents to your index](#)
- [Retrieving a specific document from your index](#)
- [Async APIs](#)
- [Create a client that can authenticate in a national cloud](#)

Querying

There are two ways to interact with the data returned from a search query.

Let's explore them with a search for a "luxury" hotel.

Use `SearchDocument` like a dictionary for search results

`SearchDocument` is the default type returned from queries when you don't provide your own. Here we perform the search, enumerate over the results, and extract data using `SearchDocument`'s dictionary indexer.

Java

```
for (SearchResult searchResult : SEARCH_CLIENT.search("luxury")) {  
    SearchDocument doc = searchResult.getDocument(SearchDocument.class);  
    String id = (String) doc.get("hotelId");  
    String name = (String) doc.get("hotelName");  
    System.out.printf("This is hotelId %s, and this is hotel name %s.%n", id, name);  
}
```

Use Java model class for search results

Define a `Hotel` class.

Java

```
public static class Hotel {
    @SimpleField(isKey = true, isFilterable = true, isSortable = true)
    private String id;
    @SearchableField(isFilterable = true, isSortable = true)
    private String name;

    public String getId() {
        return id;
    }

    public Hotel setId(String id) {
        this.id = id;
        return this;
    }

    public String getName() {
        return name;
    }

    public Hotel setName(String name) {
        this.name = name;
        return this;
    }
}
```

Use it in place of `SearchDocument` when querying.

Java

```
for (SearchResult searchResult : SEARCH_CLIENT.search("luxury")) {
    Hotel doc = searchResult.getDocument(Hotel.class);
    String id = doc.getId();
    String name = doc.getName();
    System.out.printf("This is hotelId %s, and this is hotel name %s.%n", id, name);
}
```

It is recommended, when you know the schema of the search index, to create a Java model class.

Search Options

The `SearchOptions` provide powerful control over the behavior of our queries.

Let's search for the top 5 luxury hotels with a good rating.

Java

```
SearchOptions options = new SearchOptions()
    .setFilter("rating ge 4")
    .setOrderBy("rating desc")
```

```
.setTop(5);
SearchPagedIterable searchResultsIterable = SEARCH_CLIENT.search("luxury", options,
Context.NONE);
// ...
```

Creating an index

You can use the [SearchIndexClient](#) to create a search index. Indexes can also define suggesters, lexical analyzers, and more.

There are multiple ways of preparing search fields for a search index. For basic needs, we provide a static helper method `buildSearchFields` in `SearchIndexClient` and `SearchIndexAsyncClient`, which can convert Java POJO class into `List<SearchField>`. There are three annotations `SimpleFieldProperty`, `SearchFieldProperty` and `FieldBuilderIgnore` to configure the field of model class.

Java

```
List<SearchField> searchFields = SearchIndexClient.buildSearchFields(Hotel.class,
null);
SEARCH_INDEX_CLIENT.createIndex(new SearchIndex("index", searchFields));
```

For advanced scenarios, we can build search fields using `SearchField` directly.

Java

```
List<SearchField> searchFieldList = new ArrayList<>();
searchFieldList.add(new SearchField("hotelId", SearchFieldDataType.STRING)
    .setKey(true)
    .setFilterable(true)
    .setSortable(true));

searchFieldList.add(new SearchField("hotelName", SearchFieldDataType.STRING)
    .setSearchable(true)
    .setFilterable(true)
    .setSortable(true));
searchFieldList.add(new SearchField("description", SearchFieldDataType.STRING)
    .setSearchable(true)
    .setAnalyzerName(LexicalAnalyzerName.EU_LUCENE));
searchFieldList.add(new SearchField("tags",
    SearchFieldDataType.collection(SearchFieldDataType.STRING))
    .setSearchable(true)
    .setFilterable(true)
    .setFacetable(true));
searchFieldList.add(new SearchField("address", SearchFieldDataType.COMPLEX)
    .setFields(new SearchField("streetAddress",
        SearchFieldDataType.STRING).setSearchable(true),
        new SearchField("city", SearchFieldDataType.STRING)
        .setSearchable(true))
```

```

.setFilterable(true)
.setFacetable(true)
.setSortable(true),
new SearchField("stateProvince", SearchFieldType.STRING)
.setSearchable(true)
.setFilterable(true)
.setFacetable(true)
.setSortable(true),
new SearchField("country", SearchFieldType.STRING)
.setSearchable(true)
.setFilterable(true)
.setFacetable(true)
.setSortable(true),
new SearchField("postalCode", SearchFieldType.STRING)
.setSearchable(true)
.setFilterable(true)
.setFacetable(true)
.setSortable(true)
));
// Prepare suggester.
SearchSuggester suggester = new SearchSuggester("sg",
Collections.singletonList("hotelName"));
// Prepare SearchIndex with index name and search fields.
SearchIndex index = new
SearchIndex("hotels").setFields(searchFieldList).setSuggesters(suggester);
// Create an index
SEARCH_INDEX_CLIENT.createIndex(index);

```

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

Java

```

Hotel hotel = SEARCH_CLIENT.getDocument("1", Hotel.class);
System.out.printf("This is hotelId %s, and this is hotel name %s.%n", hotel.getId(),
hotel.getName());

```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are [a few special rules for merging](#) to be aware of.

Java

```
IndexDocumentsBatch<Hotel> batch = new IndexDocumentsBatch<>();
batch.addUploadActions(Collections.singletonList(new
Hotel().setId("783").setName("Upload Inn")));
batch.addMergeActions(Collections.singletonList(new
Hotel().setId("12").setName("Renovated Ranch")));
SEARCH_CLIENT.indexDocuments(batch);
```

The request will throw `IndexBatchException` by default if any of the individual actions fail, and you can use `findFailedActionsToRetry` to retry on failed documents. There's also a `throwOnAnyError` option, and you can set it to `false` to get a successful response with an `IndexDocumentsResult` for inspection.

Async APIs

The examples so far have been using synchronous APIs, but we provide full support for async APIs as well. You'll need to use [SearchAsyncClient](#).

Java

```
SEARCH_ASYNC_CLIENT.search("luxury")
    .subscribe(result -> {
        Hotel hotel = result.getDocument(Hotel.class);
        System.out.printf("This is hotelId %s, and this is hotel name %s.%n",
            hotel.getId(), hotel.getName());
    });
}
```

Authenticate in a National Cloud

To authenticate in a [National Cloud](#), you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `audience` in `SearchClientBuilder`, `SearchIndexClientBuilder`, or `SearchIndexerClientBuilder`

Java

```
// Create a SearchClient that will authenticate through AAD in the China national
cloud.
SearchClient searchClient = new SearchClientBuilder()
    .endpoint(ENDPOINT)
    .indexName(INDEX_NAME)
    .credential(new DefaultAzureCredentialBuilder()
        .authorityHost(AzureAuthorityHosts.AZURE_CHINA)
```

```
.build()
. audience(SearchAudience.AZURE_CHINA)
.buildClient();
```

Troubleshooting

See our [troubleshooting guide](#) for details on how to diagnose various failure scenarios.

General

When you interact with Azure AI Search using this Java client library, errors returned by the service correspond to the same HTTP status codes returned for [REST API](#) requests. For example, the service will return a `404` error if you try to retrieve a document that doesn't exist in your index.

Handling Search Error Response

Any Search API operation that fails will throw an [HttpResponseException](#) with helpful [Status codes](#). Many of these errors are recoverable.

Java

```
try {
    Iterable<SearchResult> results = SEARCH_CLIENT.search("hotel");
} catch (HttpResponseException ex) {
    // The exception contains the HTTP status code and the detailed message
    // returned from the search service
    HttpResponse response = ex.getResponse();
    System.out.println("Status Code: " + response.getStatusCode());
    System.out.println("Message: " + ex.getMessage());
}
```

You can also easily [enable console logging](#) if you want to dig deeper into the requests you're making against the service.

Enabling Logging

Azure SDKs for Java provide a consistent logging story to help aid in troubleshooting application errors and expedite their resolution. The logs produced will capture the flow of an application before reaching the terminal state to help locate the root issue. View the [logging](#) wiki for guidance about enabling logging.

Default HTTP Client

By default, a Netty based HTTP client will be used. The [HTTP clients wiki](#) provides more information on configuring or changing the HTTP client.

Next steps

- Samples are explained in detail [here](#).
- Read more about the [Azure AI Search service](#)

Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a [Contributor License Agreement \(CLA\)](#) declaring that you have the right to, and actually do, grant us the rights to use your contribution.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information see the [Code of Conduct FAQ](#) or contact opencode@microsoft.com with any additional questions or comments.

Last updated on 10/10/2025

Azure AI Search client library for JavaScript - version 12.2.0

[Azure AI Search](#) (formerly known as "Azure Cognitive Search") is an AI-powered information retrieval platform that helps developers build rich search experiences and generative AI apps that combine large language models with enterprise data.

The Azure AI Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and unstructured documents. A skillset leverages APIs from Azure AI Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines and chat-style apps.

Use the `@azure/search-documents` client library to:

- Submit queries using vector, keyword, and hybrid query forms.
- Implement filtered queries for metadata, geospatial search, faceted navigation, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through semantic ranking and scoring profiles to factor in business logic or freshness.

Key links:

- [Source code ↗](#)
- [Package \(NPM\) ↗](#)
- [API reference documentation](#)
- [REST API documentation](#)
- [Product documentation](#)
- [Samples ↗](#)

Getting started

Install the `@azure/search-documents` package

Bash

```
npm install @azure/search-documents
```

Currently supported environments

- [LTS versions of Node.js](#)
- Latest versions of Safari, Chrome, Microsoft Edge, and Firefox.

See our [support policy](#) for more details.

Prerequisites

- An [Azure subscription](#)
- A [Search service](#)

To create a new search service, you can use the [Azure portal](#), [Azure PowerShell](#), or the [Azure CLI](#). Here's an example using the Azure CLI to create a free instance for getting started:

Powershell

```
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free  
--location westus
```

See [choosing a pricing tier](#) for more information about available options.

Authenticate the client

To interact with the search service, you'll need to create an instance of the appropriate client class: `SearchClient` for searching indexed documents, `SearchIndexClient` for managing indexes, or `SearchIndexerClient` for crawling data sources and loading search documents into an index. To instantiate a client object, you'll need an [endpoint](#) and [Azure roles](#) or an [API key](#). You can refer to the documentation for more information on [supported authenticating approaches](#) with the search service.

Get an API Key

An API key can be an easier approach to start with because it doesn't require pre-existing role assignments.

You can get the **endpoint** and an **API key** from the search service in the [Azure portal](#). Please refer the [documentation](#) for instructions on how to get an API key.

Alternatively, you can use the following [Azure CLI](#) command to retrieve the API key from the search service:

Powershell

```
az search admin-key show --resource-group <your-resource-group-name> --service-name <your-resource-name>
```

There are two types of keys used to access your search service: **admin (read-write)** and **query (read-only)** keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

Once you have an api-key, you can use it as follows:

ts

```
import {  
  SearchClient,  
  AzureKeyCredential,  
  SearchIndexClient,  
  SearchIndexerClient,  
} from "@azure/search-documents";  
  
// To query and manipulate documents  
const searchClient = new SearchClient(  
  "<endpoint>",  
  "<indexName>",  
  new AzureKeyCredential("<apiKey>"),  
);  
  
// To manage indexes and synonymmaps  
const indexClient = new SearchIndexClient("<endpoint>", new AzureKeyCredential(""  
  <apiKey>"));  
  
// To manage indexers, datasources and skillsets  
const indexerClient = new SearchIndexerClient("<endpoint>", new AzureKeyCredential(""  
  <apiKey>));
```

Authenticate in a National Cloud

To authenticate in a [National Cloud](#), you will need to make the following additions to your client configuration:

- Set the `Audience` in `SearchClientOptions`

```
ts
```

```
import {
  SearchClient,
  AzureKeyCredential,
  KnownSearchAudience,
  SearchIndexClient,
  SearchIndexerClient,
} from "@azure/search-documents";

// To query and manipulate documents
const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
  {
    audience: KnownSearchAudience.AzureChina,
  },
);

// To manage indexes and synonymmaps
const indexClient = new SearchIndexClient("<endpoint>", new AzureKeyCredential("",
<apiKey>"), {
  audience: KnownSearchAudience.AzureChina,
});

// To manage indexers, datasources and skillsets
const indexerClient = new SearchIndexerClient("<endpoint>", new AzureKeyCredential("",
<apiKey>"), {
  audience: KnownSearchAudience.AzureChina,
});
```

Key concepts

An Azure AI Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. (*If you're brand new to search, you can make a very rough analogy between indexes and database tables.*) The @azure/search-documents client library exposes operations on these resources through three main client types.

- `SearchClient` helps with:

- [Searching](#) your indexed documents using [vector queries](#), [keyword queries](#) and [hybrid queries](#)
- [Vector query filters](#) and [Text query filters](#)
- [Semantic ranking](#) and [scoring profiles](#) for boosting relevance
- [Autocompleting](#) partially typed search terms based on documents in the index
- [Suggesting](#) the most likely matching text in documents as a user types
- [Adding, Updating or Deleting Documents](#) documents from an index
- `SearchIndexClient` allows you to:
 - [Create, delete, update, or configure a search index](#)
 - [Declare custom synonym maps to expand or rewrite queries](#)
- `SearchIndexerClient` allows you to:
 - [Start indexers to automatically crawl data sources](#)
 - [Define AI powered Skillsets to transform and enrich your data](#)

Note: These clients cannot function in the browser because the APIs it calls do not have support for Cross-Origin Resource Sharing (CORS).

TypeScript/JavaScript specific concepts

Documents

An item stored inside a search index. The shape of this document is described in the index using the `fields` property. Each `SearchField` has a name, a datatype, and additional metadata such as if it is searchable or filterable.

Pagination

Typically you will only wish to [show a subset of search results](#) to a user at one time. To support this, you can use the `top`, `skip` and `includeTotalCount` parameters to provide a paged experience on top of search results.

Document field encoding

[Supported data types](#) in an index are mapped to JSON types in API requests/responses. The JS client library keeps these mostly the same, with some exceptions:

- `Edm.DateTimeOffset` is converted to a JS `Date`.
- `Edm.GeographyPoint` is converted to a `GeographyPoint` type exported by the client library.

- Special values of the `number` type (NaN, Infinity, -Infinity) are serialized as strings in the REST API, but are converted back to `number` by the client library.

Note: Data types are converted based on value, not the field type in the index schema. This means that if you have an ISO8601 Date string (e.g. "2020-03-06T18:48:27.896Z") as the value of a field, it will be converted to a Date regardless of how you stored it in your schema.

Examples

The following examples demonstrate the basics - please [check out our samples](#) for much more.

- [Creating an index](#)
- [Retrieving a specific document from your index](#)
- [Adding documents to your index](#)
- [Perform a search on documents](#)
 - [Querying with TypeScript](#)
 - [Querying with OData filters](#)
 - [Querying with facets](#)

Create an Index

```
ts

import { SearchIndexClient, AzureKeyCredential } from "@azure/search-documents";

const indexClient = new SearchIndexClient("<endpoint>", new AzureKeyCredential(""
<apiKey>"));

const result = await indexClient.createIndex({
  name: "example-index",
  fields: [
    {
      type: "Edm.String",
      name: "id",
      key: true,
    },
    {
      type: "Edm.Double",
      name: "awesomenessLevel",
      sortable: true,
      filterable: true,
      facetable: true,
    },
    {
      type: "Edm.String",
      name: "description",
    }
  ]
});
```

```

    searchable: true,
  },
{
  type: "Edm.ComplexType",
  name: "details",
  fields: [
    {
      type: "Collection(Edm.String)",
      name: "tags",
      searchable: true,
    },
  ],
},
{
  type: "Edm.Int32",
  name: "hiddenWeight",
  hidden: true,
},
],
});
console.log(`Index created with name ${result.name}`);

```

Retrieve a specific document from an index

A specific document can be retrieved by its primary key value:

```

ts

import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
);

const result = await searchClient.getDocument("1234");

```

Adding documents into an index

You can upload multiple documents into index inside a batch:

```

ts

import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",

```

```
new AzureKeyCredential("<apiKey>"),
);

const uploadResult = await searchClient.uploadDocuments([
// JSON objects matching the shape of the client's index
{},
{},
{},
],
);
for (const result of uploadResult.results) {
  console.log(`Uploaded ${result.key}; succeeded? ${result.succeeded}`);
}
```

Perform a search on documents

To list all results of a particular query, you can use `search` with a search string that uses [simple query syntax](#):

```
ts

import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
);

const searchResults = await searchClient.search("wifi -luxury");
for await (const result of searchResults.results) {
  console.log(result);
}
```

For a more advanced search that uses [Lucene syntax](#), specify `queryType` to be `full`:

```
ts

import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
);

const searchResults = await searchClient.search('Category:budget AND "recently
renovated"^3', {
  queryType: "full",
  searchMode: "all",
});
for await (const result of searchResults.results) {
```

```
    console.log(result);
}
```

Querying with TypeScript

In TypeScript, `SearchClient` takes a generic parameter that is the model shape of your index documents. This allows you to perform strongly typed lookup of fields returned in results. TypeScript is also able to check for fields returned when specifying a `select` parameter.

ts

```
import { SearchClient, AzureKeyCredential, SelectFields } from "@azure/search-documents";

// An example schema for documents in the index
interface Hotel {
    hotelId?: string;
    hotelName?: string | null;
    description?: string | null;
    descriptionVector?: Array<number>;
    parkingIncluded?: boolean | null;
    lastRenovationDate?: Date | null;
    rating?: number | null;
    rooms?: Array<{
        beds?: number | null;
        description?: string | null;
    }>;
}

const searchClient = new SearchClient<Hotel>(
    "<endpoint>",
    "<indexName>",
    new AzureKeyCredential("<apiKey>"),
);

const searchResults = await searchClient.search("wifi -luxury", {
    // Only fields in Hotel can be added to this array.
    // TS will complain if one is misspelled.
    select: ["hotelId", "hotelName", "rooms/beds"],
});

// These are other ways to declare the correct type for `select`.
const select = ["hotelId", "hotelName", "rooms/beds"] as const;
// This declaration lets you opt out of narrowing the TypeScript type of your
// documents,
// though the AI Search service will still only return these fields.
const selectWide: SelectFields<Hotel>[] = ["hotelId", "hotelName", "rooms/beds"];
// This is an invalid declaration. Passing this to `select` will result in a
// compiler error
// unless you opt out of including the model in the client constructor.
const selectInvalid = ["hotelId", "hotelName", "rooms/beds"];
```

```
for await (const result of searchResults.results) {
    // result.document has hotelId, hotelName, and rating.
    // Trying to access result.document.description would emit a TS error.
    console.log(result.document.hotelName);
}
```

Querying with OData filters

Using the `filter` query parameter allows you to query an index using the syntax of an [OData \\$filter expression](#).

ts

```
import { SearchClient, AzureKeyCredential, odata } from "@azure/search-documents";

const searchClient = new SearchClient(
    "<endpoint>",
    "<indexName>",
    new AzureKeyCredential("<apiKey>"),
);

const baseRateMax = 200;
const ratingMin = 4;
const searchResults = await searchClient.search("WiFi", {
    filter: odata`Rooms/any(room: room/BaseRate lt ${baseRateMax}) and Rating ge ${ratingMin}`,
    orderBy: ["Rating desc"],
    select: ["hotelId", "hotelName", "Rating"],
});
for await (const result of searchResults.results) {
    // Each result will have "HotelId", "HotelName", and "Rating"
    // in addition to the standard search result property "score"
    console.log(result);
}
```

Querying with vectors

Text embeddings can be queried using the `vector` search parameter. See [Query vectors](#) and [Filter vector queries](#) for more information.

ts

```
import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
    "<endpoint>",
    "<indexName>",
    new AzureKeyCredential("<apiKey>"),
```

```

);
const queryVector: number[] = [
  // Embedding of the query "What are the most luxurious hotels?"
];
const searchResults = await searchClient.search("*", {
  vectorSearchOptions: {
    queries: [
      {
        kind: "vector",
        vector: queryVector,
        fields: ["descriptionVector"],
        kNearestNeighborsCount: 3,
      },
    ],
  },
});
for await (const result of searchResults.results) {
  // These results are the nearest neighbors to the query vector
  console.log(result);
}

```

Querying with facets

Facets are used to help a user of your application refine a search along pre-configured dimensions. [Facet syntax](#) provides the options to sort and bucket facet values.

ts

```

import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
);

const searchResults = await searchClient.search("WiFi", {
  facets: ["category,count:3,sort:count", "rooms/baseRate,interval:100"],
});
console.log(searchResults.facets);

```

When retrieving results, a `facets` property will be available that will indicate the number of results that fall into each facet bucket. This can be used to drive refinement (e.g. issuing a follow-up search that filters on the `Rating` being greater than or equal to 3 and less than 4.)

Troubleshooting

Logging

Enabling logging can help uncover useful information about failures. In order to see a log of HTTP requests and responses, set the `AZURE_LOG_LEVEL` environment variable to `info`.

Alternatively, logging can be enabled at runtime by calling `setLogLevel` in the `@azure/logger`:

ts

```
import { setLogLevel } from "@azure/logger";  
  
setLogLevel("info");
```

For more detailed instructions on how to enable logs, you can look at the [@azure/logger package docs](#).

Next steps

- [Go further with search-documents and our samples](#)
- [Read more about the Azure AI Search service](#)

Contributing

If you'd like to contribute to this library, please read the [contributing guide](#) to learn more about how to build and test the code.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit [cla.microsoft.com](#).

This project has adopted the [Microsoft Open Source Code of Conduct](#). For more information see the [Code of Conduct FAQ](#) or contact opencode@microsoft.com with any additional questions or comments.

Related projects

- [Microsoft Azure SDK for JavaScript](#)

@azure/arm-search package

Classes

[+] [Expand table](#)

SearchManagementClient

Interfaces

[+] [Expand table](#)

AccessRule	Access rule in a network security perimeter configuration profile
AccessRuleProperties	Properties of Access Rule
AccessRuleProperties SubscriptionsItem	Subscription identifiers
AdminKeyResult	Response containing the primary and secondary admin API keys for a given Azure AI Search service.
AdminKeys	Interface representing a AdminKeys.
AdminKeysGet OptionalParams	Optional parameters.
AdminKeys RegenerateOptional Params	Optional parameters.
AsyncOperation Result	The details of a long running asynchronous shared private link resource operation.
CheckName AvailabilityInput	Input of check name availability API.
CheckName AvailabilityOutput	Output of check name availability API.
CloudError	Contains information about an API error.
CloudErrorBody	Describes a particular API error with an error code and a message.
DataPlaneAadOrApiKeyAuthOption	Indicates that either the API key or an access token from a Microsoft Entra ID tenant can be used for authentication.

DataPlaneAuthOptions	Defines the options for how the search service authenticates a data plane request. This cannot be set if 'disableLocalAuth' is set to true.
EncryptionWithCmk	Describes a policy that determines how resources within the search service are to be encrypted with customer managed keys.
Identity	Details about the search service identity. A null value indicates that the search service has no identity assigned.
IpRule	The IP restriction rule of the Azure AI Search service.
ListQueryKeysResult	Response containing the query API keys for a given Azure AI Search service.
NetworkRuleSet	Network specific rules that determine how the Azure AI Search service may be reached.
NetworkSecurityPerimeter	Information about a network security perimeter (NSP)
NetworkSecurityPerimeterConfiguration	Network security perimeter (NSP) configuration resource
NetworkSecurityPerimeterConfigurationListResult	Result of a list NSP (network security perimeter) configurations request.
NetworkSecurityPerimeterConfigurationProperties	Network security configuration properties.
NetworkSecurityPerimeterConfigurations	Interface representing a NetworkSecurityPerimeterConfigurations.
NetworkSecurityPerimeterConfigurationsGetOptionalParams	Optional parameters.
NetworkSecurityPerimeterConfigurationsListByServiceNextOptionalParams	Optional parameters.
NetworkSecurityPerimeterConfigurationsList	Optional parameters.

ByServiceOptionalParams	
NetworkSecurityPerimeterConfigurations_ReconcileHeaders	Defines headers for NetworkSecurityPerimeterConfigurations_reconcile operation.
NetworkSecurityPerimeterConfigurations_ReconcileOptionalParams	Optional parameters.
NetworkSecurityProfile	Network security perimeter configuration profile
Operation	Details of a REST API operation, returned from the Resource Provider Operations API
OperationDisplay	Localized display information for this particular operation.
OperationListResult	The result of the request to list REST API operations. It contains a list of operations and a URL to get the next set of results.
Operations	Interface representing a Operations.
OperationsListOptionalParams	Optional parameters.
PrivateEndpointConnection	Describes an existing private endpoint connection to the Azure AI Search service.
PrivateEndpointConnectionListResult	Response containing a list of private endpoint connections.
PrivateEndpointConnectionProperties	Describes the properties of an existing private endpoint connection to the search service.
PrivateEndpointConnectionPropertiesPrivateEndpoint	The private endpoint resource from Microsoft.Network provider.
PrivateEndpointConnectionPropertiesPrivateLinkService ConnectionState	Describes the current state of an existing Azure Private Link service connection to the private endpoint.
PrivateEndpointConnections	Interface representing a PrivateEndpointConnections.

PrivateEndpoint	Optional parameters.
ConnectionsDelete	
OptionalParams	
PrivateEndpoint	Optional parameters.
ConnectionsGet	
OptionalParams	
PrivateEndpoint	Optional parameters.
ConnectionsList	
ByServiceNext	
OptionalParams	
PrivateEndpoint	Optional parameters.
ConnectionsList	
ByServiceOptional	
Params	
PrivateEndpoint	Optional parameters.
ConnectionsUpdate	
OptionalParams	
PrivateLinkResource	Describes a supported private link resource for the Azure AI Search service.
PrivateLinkResourceProperties	Describes the properties of a supported private link resource for the Azure AI Search service. For a given API version, this represents the 'supported' groupIds when creating a shared private link resource.
PrivateLinkResources	Interface representing a PrivateLinkResources.
PrivateLinkResourcesListSupported	Optional parameters.
OptionalParams	
PrivateLinkResourcesResult	Response containing a list of supported Private Link Resources.
ProvisioningIssue	Describes a provisioning issue for a network security perimeter configuration
ProvisioningIssueProperties	Details of a provisioning issue for a network security perimeter (NSP) configuration. Resource providers should generate separate provisioning issue elements for each separate issue detected, and include a meaningful and distinctive description, as well as any appropriate suggestedResourceIds and suggestedAccessRules
ProxyResource	The resource model definition for a Azure Resource Manager proxy resource. It will not have tags and a location
QueryKey	Describes an API key for a given Azure AI Search service that conveys read-only permissions on the docs collection of an index.

QueryKeys	Interface representing a QueryKeys.
QueryKeysCreate OptionalParams	Optional parameters.
QueryKeysDelete OptionalParams	Optional parameters.
QueryKeysList BySearchServiceNext OptionalParams	Optional parameters.
QueryKeysList BySearchService OptionalParams	Optional parameters.
QuotaUsageResult	Describes the quota usage for a particular SKU.
QuotaUsageResult Name	The name of the SKU supported by Azure AI Search.
QuotaUsagesList Result	Response containing the quota usage information for all the supported SKUs of Azure AI Search.
Resource	Common fields that are returned in the response for all Azure Resource Manager resources
ResourceAssociation	Information about resource association
SearchManagement ClientOptionalParams	Optional parameters.
SearchManagement RequestOptions	Parameter group
SearchService	Describes an Azure AI Search service and its current state.
SearchServiceList Result	Response containing a list of Azure AI Search services.
SearchServiceUpdate	The parameters used to update an Azure AI Search service.
Services	Interface representing a Services.
ServicesCheckName AvailabilityOptionalParams	Optional parameters.
ServicesCreate OrUpdateOptionalParams	Optional parameters.

ServicesDeleteOptionalParams	Optional parameters.
ServicesGetOptionalParams	Optional parameters.
ServicesListByResourceGroupNextOptionalParams	Optional parameters.
ServicesListByResourceGroupOptionalParams	Optional parameters.
ServicesListBySubscriptionNextOptionalParams	Optional parameters.
ServicesListBySubscriptionOptionalParams	Optional parameters.
ServicesUpdateOptionalParams	Optional parameters.
ServicesUpgradeHeaders	Defines headers for Services_upgrade operation.
ServicesUpgradeOptionalParams	Optional parameters.
ShareablePrivateLinkResourceProperties	Describes the properties of a resource type that has been onboarded to private link service, supported by Azure AI Search.
ShareablePrivateLinkResourceType	Describes an resource type that has been onboarded to private link service, supported by Azure AI Search.
SharedPrivateLinkResource	Describes a shared private link resource managed by the Azure AI Search service.
SharedPrivateLinkResourceListResult	Response containing a list of shared private link resources.
SharedPrivateLinkResourceProperties	Describes the properties of an existing shared private link resource managed by the Azure AI Search service.
SharedPrivateLinkResources	Interface representing a SharedPrivateLinkResources.
SharedPrivateLinkResourcesCreate	Optional parameters.

OrUpdateOptionalParams	
SharedPrivateLinkResourcesDeleteOptionalParams	Optional parameters.
SharedPrivateLinkResourcesGetOptionalParams	Optional parameters.
SharedPrivateLinkResourcesListByServiceNextOptionalParams	Optional parameters.
SharedPrivateLinkResourcesListByServiceOptionalParams	Optional parameters.
Sku	Defines the SKU of a search service, which determines billing rate and capacity limits.
SystemData	Metadata pertaining to creation and last modification of the resource.
TrackedResource	The resource model definition for an Azure Resource Manager tracked top level resource which has 'tags' and a 'location'
UsageBySubscriptionSkuOptionalParams	Optional parameters.
Usages	Interface representing a Usages.
UsagesListBySubscriptionNextOptionalParams	Optional parameters.
UsagesListBySubscriptionOptionalParams	Optional parameters.
UserAssignedIdentity	User assigned identity properties

Type Aliases

 [Expand table](#)

AadAuthFailure	Defines values for AadAuthFailureMode.
--------------------------------	--

Mode	
AccessRuleDirection	Defines values for AccessRuleDirection. KnownAccessRuleDirection can be used interchangeably with AccessRuleDirection, this enum contains the known values that the service supports.
ActionType	Defines values for ActionType. KnownActionType can be used interchangeably with ActionType, this enum contains the known values that the service supports.
AdminKeyKind	Defines values for AdminKeyKind.
AdminKeysGetResponse	Contains response data for the get operation.
AdminKeysRegenerateResponse	Contains response data for the regenerate operation.
ComputeType	Defines values for ComputeType. KnownComputeType can be used interchangeably with ComputeType, this enum contains the known values that the service supports.
CreatedByType	Defines values for CreatedByType. KnownCreatedByType can be used interchangeably with CreatedByType, this enum contains the known values that the service supports.

HostingMode	Defines values for HostingMode.
IdentityType	<p>Defines values for IdentityType.</p> <p>KnownIdentityType can be used interchangeably with IdentityType, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>None: Indicates that any identity associated with the search service needs to be removed.</p> <p>SystemAssigned: Indicates that system-assigned identity for the search service will be enabled.</p> <p>UserAssigned: Indicates that one or more user assigned identities will be assigned to the search service.</p> <p>SystemAssigned, UserAssigned: Indicates that system-assigned identity for the search service will be enabled along with the assignment of one or more user assigned identities.</p>
IssueType	<p>Defines values for IssueType.</p> <p>KnownIssueType can be used interchangeably with IssueType, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>Unknown: Unknown issue type</p> <p>ConfigurationPropagationFailure: An error occurred while applying the network security perimeter (NSP) configuration.</p> <p>MissingPerimeterConfiguration: A network connectivity issue is happening on the resource which could be addressed either by adding new resources to the network security perimeter (NSP) or by modifying access rules.</p> <p>MissingIdentityConfiguration: A managed identity hasn't been associated with the resource. The resource will still be able to validate inbound traffic from the network security perimeter (NSP) or matching inbound access rules, but it won't be able to perform outbound access as a member of the NSP.</p>
NetworkSecurityPerimeterConfigurationProvisioningState	<p>Defines values for NetworkSecurityPerimeterConfigurationProvisioningState.</p> <p>KnownNetworkSecurityPerimeterConfigurationProvisioningState can be used interchangeably with NetworkSecurityPerimeterConfigurationProvisioningState, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>Succeeded</p> <p>Creating</p> <p>Updating</p> <p>Deleting</p> <p>Accepted</p>

	Failed Canceled
NetworkSecurityPerimeterConfigurationsGetResponse	Contains response data for the get operation.
NetworkSecurityPerimeterConfigurationsListByServiceNextResponse	Contains response data for the listByServiceNext operation.
NetworkSecurityPerimeterConfigurationsListByServiceResponse	Contains response data for the listByService operation.
NetworkSecurityPerimeterConfigurationsReconcileResponse	Contains response data for the reconcile operation.
OperationsListResponse	Contains response data for the list operation.
Origin	Defines values for Origin. KnownOrigin can be used interchangeably with Origin, this enum contains the known values that the service supports.
Known values supported by the service	
	user system user,system
PrivateEndpointConnectionsDeleteResponse	Contains response data for the delete operation.
PrivateEndpointConnectionsGetResponse	Contains response data for the get operation.
PrivateEndpointConnectionsListByServiceNextResponse	Contains response data for the listByServiceNext operation.

PrivateEndpointConnectionsListByServiceResponse	Contains response data for the listByService operation.
PrivateEndpointConnectionsUpdateResponse	Contains response data for the update operation.
PrivateLinkResourcesListSupportedResponse	Contains response data for the listSupported operation.
PrivateLinkServiceConnectionProvisioningState	Defines values for PrivateLinkServiceConnectionProvisioningState. KnownPrivateLinkServiceConnectionProvisioningState can be used interchangeably with PrivateLinkServiceConnectionProvisioningState, this enum contains the known values that the service supports.
PrivateLinkServiceConnectionStatus	Defines values for PrivateLinkServiceConnectionStatus.
ProvisioningState	Defines values for ProvisioningState.
PublicNetworkAccess	Defines values for PublicNetworkAccess. KnownPublicNetworkAccess can be used interchangeably with PublicNetworkAccess, this enum contains the known values that the service supports.
Known values supported by the service	
Updating: The private link service connection is in the process of being created along with other resources for it to be fully functional.	
Deleting: The private link service connection is in the process of being deleted.	
Failed: The private link service connection has failed to be provisioned or deleted.	
Succeeded: The private link service connection has finished provisioning and is ready for approval.	
Incomplete: Provisioning request for the private link service connection resource has been accepted but the process of creation has not commenced yet.	
Canceled: Provisioning request for the private link service connection resource has been canceled.	
Known values supported by the service	
enabled: The search service is accessible from traffic originating from the public internet.	
disabled: The search service is not accessible from traffic originating from the public internet. Access is only permitted over approved private endpoint connections.	
securedByPerimeter: The network security perimeter configuration rules allow or	

	disallow public network access to the resource. Requires an associated network security perimeter.
QueryKeysCreate Response	Contains response data for the create operation.
QueryKeysList BySearchService NextResponse	Contains response data for the listBySearchServiceNext operation.
QueryKeysList BySearchService Response	Contains response data for the listBySearchService operation.
Resource AssociationAccess Mode	Defines values for ResourceAssociationAccessMode. KnownResourceAssociationAccessMode can be used interchangeably with ResourceAssociationAccessMode, this enum contains the known values that the service supports.
<h2>Known values supported by the service</h2>	
	<p>Enforced: Enforced access mode - traffic to the resource that failed access checks is blocked</p> <p>Learning: Learning access mode - traffic to the resource is enabled for analysis but not blocked</p> <p>Audit: Audit access mode - traffic to the resource that fails access checks is logged but not blocked</p>
SearchBypass	Defines values for SearchBypass. KnownSearchBypass can be used interchangeably with SearchBypass, this enum contains the known values that the service supports.
<h2>Known values supported by the service</h2>	
	<p>None: Indicates that no origin can bypass the rules defined in the 'ipRules' section. This is the default.</p> <p>AzureServices: Indicates that requests originating from Azure trusted services can bypass the rules defined in the 'ipRules' section.</p>
SearchData Exfiltration Protection	Defines values for searchDataExfiltrationProtection. KnownSearchDataExfiltrationProtection can be used interchangeably with searchDataExfiltrationProtection, this enum contains the known values that the service supports.
<h2>Known values supported by the service</h2>	
	BlockAll: Indicates that all data exfiltration scenarios are disabled.

SearchEncryptionComplianceStatus	Defines values for SearchEncryptionComplianceStatus.
SearchEncryptionWithCmk	Defines values for SearchEncryptionWithCmk.
SearchSemanticSearch	Defines values for SearchSemanticSearch. KnownSearchSemanticSearch can be used interchangeably with SearchSemanticSearch, this enum contains the known values that the service supports.
<h2>Known values supported by the service</h2>	
	<p>disabled: Indicates that semantic reranker is disabled for the search service. This is the default.</p> <p>free: Enables semantic reranker on a search service and indicates that it is to be used within the limits of the free plan. The free plan would cap the volume of semantic ranking requests and is offered at no extra charge. This is the default for newly provisioned search services.</p> <p>standard: Enables semantic reranker on a search service as a billable feature, with higher throughput and volume of semantically reranked queries.</p>
SearchServiceStatus	Defines values for SearchServiceStatus.
ServicesCheckNameAvailabilityResponse	Contains response data for the checkNameAvailability operation.
ServicesCreateOrUpdateResponse	Contains response data for the createOrUpdate operation.
ServicesGetResponse	Contains response data for the get operation.
ServicesListByResourceGroupNextResponse	Contains response data for the listByResourceGroupNext operation.
ServicesListByResourceGroupResponse	Contains response data for the listByResourceGroup operation.
ServicesListBySubscriptionNextResponse	Contains response data for the listBySubscriptionNext operation.
ServicesListBySubscriptionResponse	Contains response data for the listBySubscription operation.

Response	
ServicesUpdateResponse	Contains response data for the update operation.
ServicesUpgradeResponse	Contains response data for the upgrade operation.
Severity	<p>Defines values for Severity. KnownSeverity can be used interchangeably with Severity, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>Warning Error</p>
SharedPrivateLinkResourceAsyncResult	<p>Defines values for SharedPrivateLinkResourceAsyncResult. KnownSharedPrivateLinkResourceAsyncResult can be used interchangeably with SharedPrivateLinkResourceAsyncResult, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>Running Succeeded Failed</p>
SharedPrivateLinkResourceProvisioningState	<p>Defines values for SharedPrivateLinkResourceProvisioningState. KnownSharedPrivateLinkResourceProvisioningState can be used interchangeably with SharedPrivateLinkResourceProvisioningState, this enum contains the known values that the service supports.</p>
<h2>Known values supported by the service</h2>	
	<p>Updating: The shared private link resource is in the process of being created along with other resources for it to be fully functional. Deleting: The shared private link resource is in the process of being deleted. Failed: The shared private link resource has failed to be provisioned or deleted. Succeeded: The shared private link resource has finished provisioning and is ready for approval. Incomplete: Provisioning request for the shared private link resource has been accepted but the process of creation has not commenced yet.</p>
SharedPrivateLinkResourceStatus	<p>Defines values for SharedPrivateLinkResourceStatus. KnownSharedPrivateLinkResourceStatus can be used interchangeably with SharedPrivateLinkResourceStatus, this enum contains the known values that the service supports.</p>

Known values supported by the service

Pending: The shared private link resource has been created and is pending approval.

Approved: The shared private link resource is approved and is ready for use.

Rejected: The shared private link resource has been rejected and cannot be used.

Disconnected: The shared private link resource has been removed from the service.

SharedPrivateLinkResourcesCreateOrUpdateResponse	Contains response data for the createOrUpdate operation.
--	--

SharedPrivateLinkResourcesGetResponse	Contains response data for the get operation.
---	---

SharedPrivateLinkResourcesListByServiceNextResponse	Contains response data for the listByServiceNext operation.
---	---

SharedPrivateLinkResourcesListByServiceResponse	Contains response data for the listByService operation.
---	---

SkuName	Defines values for SkuName. KnownSkuName can be used interchangeably with SkuName, this enum contains the known values that the service supports.
-------------------------	--

Known values supported by the service

free: Free tier, with no SLA guarantees and a subset of the features offered on billable tiers.

basic: Billable tier for a dedicated service having up to 3 replicas.

standard: Billable tier for a dedicated service having up to 12 partitions and 12 replicas.

standard2: Similar to 'standard', but with more capacity per search unit.

standard3: The largest Standard offering with up to 12 partitions and 12 replicas (or up to 3 partitions with more indexes if you also set the hostingMode property to 'highDensity').

storage_optimized_I1: Billable tier for a dedicated service that supports 1TB per partition, up to 12 partitions.

storage_optimized_I2: Billable tier for a dedicated service that supports 2TB per partition, up to 12 partitions.

UnavailableNameReason	Defines values for UnavailableNameReason. KnownUnavailableNameReason can be used interchangeably with
---------------------------------------	--

`UnavailableNameReason`, this enum contains the known values that the service supports.

Known values supported by the service

Invalid: The search service name doesn't match naming requirements.

AlreadyExists: The search service name is already assigned to a different search service.

UpgradeAvailable	Defines values for <code>UpgradeAvailable</code> . <code>KnownUpgradeAvailable</code> can be used interchangeably with <code>UpgradeAvailable</code> , this enum contains the known values that the service supports.
----------------------------------	--

Known values supported by the service

notAvailable: An upgrade is currently not available for the service.

available: There is an upgrade available for the service.

UsageBySubscriptionSkuResponse	Contains response data for the <code>usageBySubscriptionSku</code> operation.
--	---

UsagesListBySubscriptionNextResponse	Contains response data for the <code>listBySubscriptionNext</code> operation.
--	---

UsagesListBySubscriptionResponse	Contains response data for the <code>listBySubscription</code> operation.
--	---

Enums

[] [Expand table](#)

KnownAccessRuleDirection	Known values of <code>AccessRuleDirection</code> that the service accepts.
KnownActionType	Known values of <code>ActionType</code> that the service accepts.
KnownComputeType	Known values of <code>ComputeType</code> that the service accepts.
KnownCreatedByType	Known values of <code>CreatedByType</code> that the service accepts.
KnownIdentityType	Known values of <code>IdentityType</code> that the service accepts.
KnownIssueType	Known values of <code>IssueType</code> that the service accepts.
KnownNetworkSecurityPerimeterConfiguration	Known values of <code>NetworkSecurityPerimeterConfigurationProvisioningState</code> that the service supports.

ProvisioningState	service accepts.
KnownOrigin	Known values of Origin that the service accepts.
KnownPrivateLinkServiceConnectionProvisioningState	Known values of PrivateLinkServiceConnectionProvisioningState that the service accepts.
KnownPublicNetworkAccess	Known values of PublicNetworkAccess that the service accepts.
KnownResourceAssociationAccessMode	Known values of ResourceAssociationAccessMode that the service accepts.
KnownSearchBypass	Known values of SearchBypass that the service accepts.
KnownSearchDataExfiltrationProtection	Known values of SearchDataExfiltrationProtection that the service accepts.
KnownSearchSemanticSearch	Known values of SearchSemanticSearch that the service accepts.
KnownSeverity	Known values of Severity that the service accepts.
KnownSharedPrivateLinkResourceAsyncResult	Known values of SharedPrivateLinkResourceAsyncResult that the service accepts.
KnownSharedPrivateLinkResourceProvisioningState	Known values of SharedPrivateLinkResourceProvisioningState that the service accepts.
KnownSharedPrivateLinkResourceStatus	Known values of SharedPrivateLinkResourceStatus that the service accepts.
KnownSkuName	Known values of SkuName that the service accepts.
KnownUnavailableNameReason	Known values of UnavailableNameReason that the service accepts.
KnownUpgradeAvailable	Known values of UpgradeAvailable that the service accepts.

Functions

[\[\]](#) [Expand table](#)

getContinuationToken(unknown)	Given the last <code>.value</code> produced by the <code>byPage</code> iterator, returns a continuation token that can be used to begin paging from that point later.
-------------------------------	---

Function Details

getContinuationToken(unknown)

Given the last `.value` produced by the `byPage` iterator, returns a continuation token that can be used to begin paging from that point later.

TypeScript

```
function getContinuationToken(page: unknown): string | undefined
```

Parameters

page `unknown`

An object from accessing `value` on the `IteratorResult` from a `byPage` iterator.

Returns

`string | undefined`

The continuation token that can be passed into `byPage()` during future calls.

Azure AI Search client library for Python - version 11.6.0

[Azure AI Search](#) (formerly known as "Azure Cognitive Search") is an AI-powered information retrieval platform that helps developers build rich search experiences and generative AI apps that combine large language models with enterprise data.

Azure AI Search is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and unstructured documents. A skillset leverages APIs from Azure AI Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines and chat-style apps.

Use the `Azure.Search.Documents` client library to:

- Submit queries using vector, keyword, and hybrid query forms.
- Implement filtered queries for metadata, geospatial search, faceted navigation, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through semantic ranking and scoring profiles to factor in business logic or freshness.

[Source code ↗](#) | [Package \(PyPI\) ↗](#) | [Package \(Conda\) ↗](#) | [API reference documentation ↗](#) |
[Product documentation](#) | [Samples ↗](#)

Getting started

Install the package

Install the Azure AI Search client library for Python with [pip ↗](#):

Bash

```
pip install azure-search-documents
```

Prerequisites

- Python 3.8 or later is required to use this package.
- You need an [Azure subscription](#) and an [Azure AI Search service](#) to use this package.

To create a new search service, you can use the [Azure portal](#), [Azure PowerShell](#), or the [Azure CLI](#).

Powershell

```
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free  
--location westus
```

See [choosing a pricing tier](#) for more information about available options.

Authenticate the client

To interact with the search service, you'll need to create an instance of the appropriate client class: `SearchClient` for searching indexed documents, `SearchIndexClient` for managing indexes, or `SearchIndexerClient` for crawling data sources and loading search documents into an index. To instantiate a client object, you'll need an `endpoint` and [Azure roles](#) or an [API key](#). You can refer to the documentation for more information on [supported authenticating approaches](#) with the search service.

Get an API Key

An API key can be an easier approach to start with because it doesn't require pre-existing role assignments.

You can get the `endpoint` and an `API key` from the Search service in the [Azure portal](#). Please refer the [documentation](#) for instructions on how to get an API key.

Alternatively, you can use the following [Azure CLI](#) command to retrieve the API key from the Search service:

Powershell

```
az search admin-key show --service-name <mysearch> --resource-group <mysearch-rg>
```

There are two types of keys used to access your search service: **admin** (*read-write*) and **query** (*read-only*) keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

Create a SearchClient

To instantiate the `SearchClient`, you'll need the **endpoint**, **API key** and **index name**:

Python

```
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

service_endpoint = os.environ["AZURE_SEARCH_SERVICE_ENDPOINT"]
index_name = os.environ["AZURE_SEARCH_INDEX_NAME"]
key = os.environ["AZURE_SEARCH_API_KEY"]

search_client = SearchClient(service_endpoint, index_name, AzureKeyCredential(key))
```

Create a client using Microsoft Entra ID authentication

You can also create a `SearchClient`, `SearchIndexClient`, or `SearchIndexerClient` using Microsoft Entra ID authentication. Your user or service principal must be assigned the "Search Index Data Reader" role. Using the [DefaultAzureCredential](#) you can authenticate a service using Managed Identity or a service principal, authenticate as a developer working on an application, and more all without changing code. Please refer the [documentation](#) for instructions on how to connect to Azure AI Search using Azure role-based access control (Azure RBAC).

Before you can use the `DefaultAzureCredential`, or any credential type from [Azure.Identity](#), you'll first need to [install the Azure.Identity package](#).

To use `DefaultAzureCredential` with a client ID and secret, you'll need to set the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables; alternatively, you can pass those values to the `ClientSecretCredential` also in `Azure.Identity`.

Make sure you use the right namespace for `DefaultAzureCredential` at the top of your source file:

Python

```
from azure.identity import DefaultAzureCredential
from azure.search.documents import SearchClient

service_endpoint = os.getenv("AZURE_SEARCH_SERVICE_ENDPOINT")
index_name = os.getenv("AZURE_SEARCH_INDEX_NAME")
credential = DefaultAzureCredential()

search_client = SearchClient(service_endpoint, index_name, credential)
```

Key concepts

An Azure AI Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. (*If you're brand new to search, you can make a very rough analogy between indexes and database tables.*) The Azure.Search.Documents client library exposes operations on these resources through three main client types.

- `SearchClient` helps with:
 - [Searching](#) your indexed documents using [vector queries](#), [keyword queries](#) and [hybrid queries](#)
 - [Vector query filters](#) and [Text query filters](#)
 - [Semantic ranking](#) and [scoring profiles](#) for boosting relevance
 - [Autocompleting](#) partially typed search terms based on documents in the index
 - [Suggesting](#) the most likely matching text in documents as a user types
 - [Adding, Updating or Deleting Documents](#) documents from an index
- `SearchIndexClient` allows you to:
 - [Create, delete, update, or configure a search index](#)
 - [Declare custom synonym maps to expand or rewrite queries](#)
- `SearchIndexerClient` allows you to:
 - [Start indexers to automatically crawl data sources](#)
 - [Define AI powered Skillsets to transform and enrich your data](#)

Azure AI Search provides two powerful features: **semantic ranking** and **vector search**.

Semantic ranking enhances the quality of search results for text-based queries. By enabling semantic ranking on your search service, you can improve the relevance of search results in two ways:

- It applies secondary ranking to the initial result set, promoting the most semantically relevant results to the top.

- It extracts and returns captions and answers in the response, which can be displayed on a search page to enhance the user's search experience.

To learn more about semantic ranking, you can refer to the [documentation](#).

Vector search is an information retrieval technique that uses numeric representations of searchable documents and query strings. By searching for numeric representations of content that are most similar to the numeric query, vector search can find relevant matches, even if the exact terms of the query are not present in the index. Moreover, vector search can be applied to various types of content, including images and videos and translated text, not just same-language text.

To learn how to index vector fields and perform vector search, you can refer to the [sample](#). This sample provides detailed guidance on indexing vector fields and demonstrates how to perform vector search.

Additionally, for more comprehensive information about vector search, including its concepts and usage, you can refer to the [documentation](#). The documentation provides in-depth explanations and guidance on leveraging the power of vector search in Azure AI Search.

The `Azure.Search.Documents` client library (v1) provides APIs for data plane operations. The previous `Microsoft.Azure.Search` client library (v10) is now retired. It has many similar looking APIs, so please be careful to avoid confusion when exploring online resources. A good rule of thumb is to check for the namespace `Azure.Search.Documents`; when you're looking for API reference.

Examples

The following examples all use a simple [Hotel data set](#) that you can [import into your own index from the Azure portal](#). These are just a few of the basics - please [check out our Samples](#) for much more.

- [Querying](#)
- [Creating an index](#)
- [Adding documents to your index](#)
- [Retrieving a specific document from your index](#)
- [Async APIs](#)

Querying

Let's start by importing our namespaces.

Python

```
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
```

We'll then create a `SearchClient` to access our hotels search index.

Python

```
index_name = "hotels"
# Get the service endpoint and API key from the environment
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

# Create a client
credential = AzureKeyCredential(key)
client = SearchClient(endpoint=endpoint,
                      index_name=index_name,
                      credential=credential)
```

Let's search for a "luxury" hotel.

Python

```
results = client.search(search_text="luxury")

for result in results:
    print("{}: {}".format(result["hotelId"], result["hotelName"]))
```

Creating an index

You can use the `SearchIndexClient` to create a search index. Fields can be defined using convenient `SimpleField`, `SearchableField`, or `ComplexField` models. Indexes can also define suggesters, lexical analyzers, and more.

Python

```
client = SearchIndexClient(service_endpoint, AzureKeyCredential(key))
name = "hotels"
fields = [
    SimpleField(name="hotelId", type=SearchFieldDataType.String, key=True),
    SimpleField(name="baseRate", type=SearchFieldDataType.Double),
    SearchableField(name="description", type=SearchFieldDataType.String,
collection=True),
    ComplexField(
        name="address",
        fields=[
            SimpleField(name="streetAddress", type=SearchFieldDataType.String),
```

```
        SimpleField(name="city", type=SearchFieldDataType.String),
    ],
    collection=True,
),
]
cors_options = CorsOptions(allowed_origins=[ "*"], max_age_in_seconds=60)
scoring_profiles: List[ScoringProfile] = []
index = SearchIndex(name=name, fields=fields, scoring_profiles=scoring_profiles,
cors_options=cors_options)

result = client.create_index(index)
```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are [a few special rules for merging](#) to be aware of.

Python

```
DOCUMENT = {
    "category": "Hotel",
    "hotelId": "1000",
    "rating": 4.0,
    "rooms": [],
    "hotelName": "Azure Inn",
}

result = search_client.upload_documents(documents=[DOCUMENT])

print("Upload of new document succeeded: {}".format(result[0].succeeded))
```

Authenticate in a National Cloud

To authenticate in a [National Cloud](#), you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `audience` in `SearchClient`, `SearchIndexClient`, or `SearchIndexerClient`

Python

```
# Create a SearchClient that will authenticate through AAD in the China national
cloud.
import os
from azure.identity import DefaultAzureCredential, AzureAuthorityHosts
from azure.search.documents import SearchClient
```

```
index_name = "hotels"
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]
credential = DefaultAzureCredential(authority=AzureAuthorityHosts.AZURE_CHINA)

search_client = SearchClient(endpoint, index_name, credential=credential,
audience="https://search.azure.cn")
```

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

Python

```
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

search_client = SearchClient(service_endpoint, index_name, AzureKeyCredential(key))

result = search_client.get_document(key="23")

print("Details for hotel '23' are:")
print("    Name: {}".format(result["hotelName"]))
print("    Rating: {}".format(result["rating"]))
print("    Category: {}".format(result["category"]))
```

Async APIs

This library includes a complete async API. To use it, you must first install an async transport, such as [aiohttp](#). See [azure-core documentation](#) for more information.

Python

```
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.aio import SearchClient

search_client = SearchClient(service_endpoint, index_name, AzureKeyCredential(key))

async with search_client:
    results = await search_client.search(search_text="spa")

    print("Hotels containing 'spa' in the name (or other fields):")
    async for result in results:
        print("    Name: {} (rating {})".format(result["hotelName"],
result["rating"]))
```

Troubleshooting

General

The Azure AI Search client will raise exceptions defined in [Azure Core](#).

Logging

This library uses the standard [logging](#) library for logging. Basic information about HTTP sessions (URLs, headers, etc.) is logged at INFO level.

Detailed DEBUG level logging, including request/response bodies and unredacted headers, can be enabled on a client with the `logging_enable` keyword argument:

Python

```
import sys
import logging
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

# Create a logger for the 'azure' SDK
logger = logging.getLogger('azure')
logger.setLevel(logging.DEBUG)

# Configure a console output
handler = logging.StreamHandler(stream=sys.stdout)
logger.addHandler(handler)

# This client will log detailed information about its HTTP sessions, at DEBUG level
client = SearchClient("<service endpoint>", "<index_name>", AzureKeyCredential("<api key>"), logging_enable=True)
```

Similarly, `logging_enable` can enable detailed logging for a single operation, even when it isn't enabled for the client:

Python

```
result = client.search(search_text="spa", logging_enable=True)
```

Next steps

- Go further with `Azure.Search.Documents` and our https://github.com/Azure/azure-sdk-for-python/blob/azure-search-documents_11.6.0/sdk/search/azure-search-

[documents/samples ↗](#)

- Read more about the [Azure AI Search service](#)

Contributing

See our [Search CONTRIBUTING.md ↗](#) for details on building, testing, and contributing to this library.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit [cla.microsoft.com ↗](#).

This project has adopted the [Microsoft Open Source Code of Conduct ↗](#). For more information, see the [Code of Conduct FAQ ↗](#) or contact opencode@microsoft.com with any additional questions or comments.



Related projects

- [Microsoft Azure SDK for Python ↗](#)



Last updated on 10/09/2025

search Package

Packages

 [Expand table](#)

[aio](#)

[models](#)

[operations](#)

Classes

 [Expand table](#)

[SearchManagementClient](#) Client that can be used to manage Azure AI Search services and API keys.

Supported data types (Azure AI Search)

10/08/2025

This article describes the data types supported by Azure AI Search. Fields and the values used in filter expressions are typed according to the [Entity Data Model \(EDM\)](#). Specifying an EDM data type is a requirement for field definition.

! Note

If you're using [indexers](#), see [Data type map for indexers in Azure AI Search](#) for more information about how indexers map source-specific data types to EDM data types in a search index.

EDM data types for vector fields

A [vector field](#) type must be valid for the output of your embedding model. For example, if you use text-embedding-ada-002, the output format is `Float32` or `Collection(Edm.Single)`. In this scenario, you can't assign an `Int8` data type because casting from `float` to `int` primitives is prohibited. However, you can cast from `Float32` to `Float16` or `(Collection(Edm.Half))`.

Vector fields are an array of embeddings. In EDM, an array is a collection.

[] [Expand table](#)

Data type	Vector type	Description	Recommended use
<code>Collection(Edm.Byte)</code>	Binary	1-bit unsigned binary. Generally available in Create or Update Index .	Supports integration with models that emit binary embeddings, such as Cohere's v3 binary embedding models or custom quantization logic that emits 1-bit unsigned binary output. For fields of type <code>Collection(Edm.Byte)</code> , see Index binary data for help with specifying the field definition and vector search algorithms for binary data.
<code>Collection(Edm.Single)</code>	<code>Float32</code>	32-bit floating point. Generally available in Create or Update Index .	Default data type in Microsoft tools that create vector fields on your behalf. Strikes a balance between precision and efficiency. Most embedding models emit vectors as <code>Float32</code> .

Data type	Vector type	Description	Recommended use
Collection(Edm.Half)	Float16	16-bit floating point with lower precision and range. Generally available in Create or Update Index .	Useful for scenarios where memory and computational efficiency are critical, and where sacrificing some precision is acceptable. Often leads to faster query times and reduced memory footprint compared to <code>Float32</code> , although with slightly reduced accuracy. You can assign a <code>Float16</code> type to index <code>Float32</code> embeddings as <code>Float16</code> . You can also use <code>Float16</code> for embedding models or custom quantization processes that emit <code>Float16</code> natively.
Collection(Edm.Int16)	Int16	16-bit signed integer. Generally available in Create or Update Index .	Offers reduced memory footprint compared to <code>Float32</code> and support for higher-precision quantization methods while still retaining sufficient precision for many applications. Suitable for cases where memory efficiency is important. Requires that you have custom quantization that outputs vectors as <code>Int16</code> .
Collection(Edm.SByte)	Int8	8-bit signed integer. Generally available in Create or Update Index .	Provides significant memory and computational efficiency gains compared to <code>Float32</code> or <code>Float16</code> . However, it likely requires supplemental techniques (like quantization and oversampling) to offset the reduction in precision and recall appropriately. Requires that you have custom quantization that outputs vectors as <code>Int8</code> .

EDM data types for nonvector fields

[] [Expand table](#)

Data type	Description
Edm.String	Text data.
Edm.Boolean	Contains true/false values.
Edm.Int32	32-bit integer values.
Edm.Int64	64-bit integer values.
Edm.Double	Double-precision IEEE 754 floating-point values.

Data type	Description
<code>Edm.DateTimeOffset</code>	Date and time values represented in the OData V4 format: <code>yyyy-MM-ddTHH:mm:ss.fffZ</code> or <code>yyyy-MM-ddTHH:mm:ss.fff[+ -]HH:mm</code> . Precision of <code>DateTimeOffset</code> fields is limited to milliseconds. If you upload <code>DateTimeOffset</code> values with submillisecond precision, the value returned is rounded up to milliseconds (for example, <code>2024-04-15T10:30:09.7552052Z</code> is returned as <code>2024-04-15T10:30:09.7550000Z</code>). When you upload <code>DateTimeOffset</code> values with time zone information to your index, Azure AI Search normalizes these values to UTC. For example, <code>2024-01-13T14:03:00-08:00</code> is stored as <code>2024-01-13T22:03:00Z</code> . If you need to store time zone information, add an extra field to your index.
<code>Edm.GeographyPoint</code>	A point representing a geographic location on the globe. For request and response bodies, the representation of values of this type follows the GeoJSON "Point" type format. For URLs, OData uses a literal form based on the WKT standard. A point literal is constructed as <code>geography'POINT(lon lat)'</code> .
<code>Edm.ComplexType</code>	Objects whose properties map to subfields that can be of any other supported data type. This type enables indexing of structured hierarchical data such as JSON. Objects in a field of type <code>Edm.ComplexType</code> can contain nested objects, but the level of nesting is limited. The limits are described in Service limits .
<code>Collection(Edm.String)</code>	A list of strings.
<code>Collection(Edm.Boolean)</code>	A list of boolean values.
<code>Collection(Edm.Int32)</code>	A list of 32-bit integer values.
<code>Collection(Edm.Int64)</code>	A list of 64-bit integer values.
<code>Collection(Edm.Double)</code>	A list of double-precision numeric values.
<code>Collection(Edm.DateTimeOffset)</code>	A list of date time values.
<code>Collection(Edm.GeographyPoint)</code>	A list of points representing geographic locations.
<code>Collection(Edm.ComplexType)</code>	A list of objects of type <code>Edm.ComplexType</code> . There's a limit on the maximum number of elements across all collections of type <code>Edm.ComplexType</code> in a document. See Service limits for details.

All of the above types are nullable, except for collections of primitive and complex types, for example, `Collection(Edm.String)`. Nullable fields can be explicitly set to null. They're automatically set to null when omitted from a document that is uploaded to an Azure AI Search index. Collection fields are automatically set to empty (`[]` in JSON) when they're omitted from a document. Also, it isn't possible to store a null value in a collection field.

Unlike complex collections, there's no upper limit specifically on the number of items in a collection of primitive types, but the [16-MB upper limit on payload size](#) applies to all parts of documents, including collections.

Geospatial data type used in filter expressions

In Azure AI Search, geospatial search is expressed as a filter.

Edm.GeographyPolygon is a polygon representing a geographic region on the globe. While this type can't be used in document fields, it can be used as an argument to the `geo.intersects` function. The literal form for URLs in OData is based on the [WKT \(Well-known text\)](#) and [OGC's simple feature access standards](#). A polygon literal is constructed as `geography'POLYGON((lon lat, lon lat, ...))'`.

Important

Points in a polygon *must* be in counterclockwise order. Points in a polygon are interpreted in counterclockwise order, relative to the inside of the polygon. For example, a 4-point closed polygon around London would be -0.3°W 51.6°N [top left] , -0.3°W 51.4°N [bottom left], 0.1°E 51.4°N [bottom right], 0.1°E 51.6°N [top right], -0.3°W 51.6°N [starting point].

See also

- [Indexer overview](#)
- [Creating indexers](#)
- [Data sources gallery](#)

Data type map for indexers (Azure AI Search)

10/08/2025

When you're building an index schema for indexer-based indexing, the data types in source data must map to an allowed data type for the fields in the target index.

This article provides data type comparisons between SQL Data Types, JSON data types, and Azure AI Search. It contains the following sections:

- [SQL Server Data Types to Azure AI Search Data Types](#)
- [JSON Data Types to Azure AI Search Data Types](#)

SQL Server Data Types to Azure AI Search Data Types

[Expand table](#)

SQL Server Data Type	Allowed target index field types	Notes
bit	Edm.Boolean, Edm.String	
int, smallint, tinyint	Edm.Int32, Edm.Int64, Edm.String	
bigint	Edm.Int64, Edm.String	
real, float	Edm.Double, Edm.String	
smallmoney, money	Edm.String	Azure AI Search doesn't support converting decimal types into Edm.Double because doing so would lose precision.
decimal		
numeric		
char, nchar, varchar, nvarchar	Edm.String	
		Collection(Edm.String). See Field Mapping Functions for details on how to transform a string column into a Collection(Edm.String)
smalldatetime, datetime,	Edm.DateTimeOffset, Edm.String	

SQL Server Data Type	Allowed target index field types	Notes
datetime2, date, datetimeoffset		
uniqueidentifier	Edm.String	
rowversion	N/A	Row-version columns can't be stored in the search index, but they can be used for change tracking.
geography	Edm.GeographyPoint, Edm.String	If using geography data types , only geography instances of type POINT with SRID 4326 (which is the default) are supported. If using strings, only GeoJSON points in the following format are supported: <code>{"type": "Point", "coordinates": [long, lat]}</code>
time, timespan	N/A	Not supported.
varbinary		
image		
xml		
geometry		
CLR types		

JSON Data Types to Azure AI Search Data Types

[Expand table](#)

JSON data type	Allowed target index field types
bool	Edm.Boolean, Edm.String
Integral numbers	Edm.Int32, Edm.Int64, Edm.String
Floating-point numbers	Edm.Double, Edm.String
string	Edm.String
arrays of primitive types, for example ["a", "b", "c"]	Collection(Edm.String)

JSON data type	Allowed target index field types
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON point objects	Edm.GeographyPoint GeoJSON points are JSON objects in the following format : <code>{"type": "Point", "coordinates": [long, lat]}</code>
JSON objects	Edm.ComplexType Azure AI Search maps JSON objects to corresponding complex type schemas

See also

- [Supported data types](#)
- [Azure AI Search REST APIs](#)

Stopwords reference (Microsoft analyzers)

Article • 08/28/2024

When text is indexed into Azure AI Search, it's processed by analyzers so it can be efficiently stored in a search index. During this [lexical analysis](#) process, [language analyzers](#) will remove stopwords specific to that language. Stopwords are non-essential words such as "the" or "an" that can be removed without compromising the lexical integrity of your content.

Stopword removal applies to all supported [Lucene and Microsoft analyzers](#) used in Azure AI Search.

This article lists the stopwords used by the Microsoft analyzer for each language.

For the stopword list for Lucene analyzers, see the [Apache Lucene source code on GitHub](#).

Tip

To view the output of any given analyzer, call the [Analyze Text REST API](#). This API is often helpful for debugging unexpected search results.

Arabic (ar.microsoft)

Bengali (bn.microsoft)

в о б и н е на ч то по д л я как от э то из за толь ко и ли их в се е го он но до же то
так у же а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ѿ ѿ ѿ
я

Bulgarian (bg.microsoft)

вж до е и на от то у че

Catalan (ca.microsoft)

la el l les de d del dels i un una uns unes a als al en és es s se hi

Chinese Simplified (zh-Hans.microsoft)

?about \$ 1 2 3 4 5 6 7 8 9 0 _ a b c d e f g h i j k l m n o p q r s t u v w x y z
after all also an and another any are as at be because been before being
between both but by came can come could did do each for from get got had has
have he her here him himself his how if in into is it like make many me might
more most much must my never now of on only or other our out over said same
see should since some still such take than that the their them then there
these they this those through to too under up very was way we well were what
where which while who with would you your 的 在 了 是 有 为 这 我 也 就 他 与
等 以 着 而 从 并 还 已 但 你 之 更 又 得 她 它 很 其 该 那 各

Chinese Traditional (zh-Hant.microsoft)

?about \$ 1 2 3 4 5 6 7 8 9 0 _ a b c d e f g h i j k l m n o p q r s t u v w x y z
after all also an and another any are as at be because been before being
between both but by came can come could did do each for from get got had has
have he her here him himself his how if in into is it like make many me might
more most much must my never now of on only or other our out over said same
see should since some still such take than that the their them then there
these they this those through to too under up very was way we well were what
where which while who with would you your 的 一 不 在 人 有 是 為 以 於 上 他
而 後 之 來 及 了 因 下 可 到 由 這 與 也 此 但 並 個 其 已 無 小 我 們 起 最
再 今 去 好 只 又 或 很 亦 某 把 那 你 乃 它

Croatian (hr.microsoft)

i u je se na za da su o od a s

Czech (cs.microsoft)

na se je že do to ve ale za si pro po by od už který bude jako tak jsou jsem
jsme však podle až jen ze před také jeho má když byl co jak nebo při ještě
aby než budou ani jaké další kteří není bylo mezi v a i ač ačkoli přece no ne
ano která které kterou kterými budu budeme budete byli byly o ať Á á Ě ě É é
Í í Ó ó Ú ú Ū ū Ď ď Ň ň Č č Ř ř Š ř Ŧ ŭ Ž ź Ÿ Ÿ a b c d e f g h i j k l
m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V
W X Y Z 0 1 2 3 4 5 6 7 8 9 I II III IV V VI VII VIII IX X XI XII XIII XIV
XVI XVII XX X I V L D M C LD CCC IV VII XXXII LXXI VIII DCII LXX DCII. III
DCCLII CDIX XXVII LIV MMVI MCMLXXII LXX DCC LII DCCIV DCV LXXIV LXV IXX XVIII

Danish (da.microsoft)

af alting at begge countries de dem den dén denne der deres det dét dette dig
din dine disse dit du eder eders en én enhver ens er et ét ethvert for fra
ham han hans har hende hendes hin hinanden hun hverandre hvis hvo I ikke ingen
ingenting jeg jer jeres man med men mig min mine mit nogen nogle og om os på
sig sin sine sit som somme til være vi vor Vor vore vores

Dutch (nl.microsoft)

de van en het in een is zijn de met op te voor die door dat aan tot als of
in hij werd het uit bij ook niet wordt worden was er naar om zich maar heeft
dan over deze nog meer kan ze hebben hun onder een kunnen tussen tegen dit na
hij andere al zij veel men geen werden wel waar zie vooral weer deel je wat
nu ten alle op van had waren maar moet zo zeer hem bij ook

English (en.microsoft)

is and in it of the to that this these those is was for on be with as by at
have are this not but had from or I my me mine myself you your yours yourself
he him his himself she her hers herself it its itself we our ours ourselves

they them their theirs themselves A B C D E F G H J K L M N O P Q R S T U
V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

Finnish (fi.microsoft)

ai ainoa ainoaa ainoaan ainoaksi ainoalla ainoalle ainoalta ainoan ainoassa
ainoasta ali alitse alla alle alta edellä edelle edeltä ehkä ei enemmän eniten
ennen entä entäs erääkseen erääksi eräällä eräälle eräältä eräään eräässä
eräästä eräs eräskin erästä että hän häneen häneksi hänellä hänelle häneltä
hänen hänessä hänestä harva harvaa harvaan harvaksi harvalla harvalle harvalta
harvan harvassa harvasta harvat harvoihin harvoiksi harvoilla harvoille
harvoilta harvoissa harvoista harvoja harvojen he heidän heihin heiksi heillä
heille heiltä heissä heistä hiukan huolimatta ilman itse itseäni itseeni
itseensä itsekseeni itselläni itselleni itseni itsessäni itsestäni ja jälkeen
johon johonkin joiden joidenkin joihin joihinkin joiksikin joilla joillakin
joille joillekin joilta joiltakin joissa joissakin joista joistakin joita joka
jokainen jokaiseksi jokaisella jokaiselle jokaiselta jokaisen jokaisessa
jokaisesta jokaista jokin joko joksikin jokunen jalaiseen jolla jollainen
jollaiseen jollaiseksi jollaisella jollaiselle jollaiselta jollaisen jollaisessa
jollaisesta jollaiset jollaisia jollaisiin jollaisiksi jollaisilla jollaisille
jollaisilta jollaisissa jollaisista jollaista jollaisten jollakin jolle jollekin
jolta jonka jonkin jonkinlainen jonkinlaiseen jonkinlaiseksi jonkinlaisella
jonkinlaiselle jonkinlaiselta jonkinlaisen jonkinlaisessa jonkinlaisesta
jonkinlaiset jonkinlaisia jonkinlaisiin jonkinlaisiksi jonkinlaisilla
jonkinlaisille jonkinlaisilta jonkinlaisissa jonkinlaisista jonkinlaista
jonkinlaisten jos jossa jossakin josta jostakin jota jotakin jotka jotkin jotta
kaikki kanssa kauas kaukana kautta keiden keitä kenä kera keskellä keskelle
ketä ketkä kohti koska kuhunkin kuin kuinka kuitenkin kuka kukin kullakin
kullein kuluessa kuluttua kummallakin kummallekin kummaltakin kummanakin
kummassakin kummastakin kummatkin kumpaakin kumpaankin kumpanakin kumpiakin
kumpienkin kumpikin kun kunkin kussakin kutakin kyllä kylläkin lähellä lähelle
läpi lisäksi lukuunottamatta luokse luona luota me meidän meihin meiksi meillä
meille meiltä meissä meistä meitä mikä miksei miksi milloin minä minua minuksi
minulla minulle minulta minun minussa minusta minuun mistä miten mitkä molemmat
molemmiksi molemmilla molemmille molemmissa molemmista molempia molempien
molempien moneen moneksi monella monelle monelta monen monenlainen monenlaiseen
monenlaiseksi monenlaisella monenlaiselle monenlaiselta monenlaisen monenlaisessa

monenlaisesta monenlaiset monenlaisia monenlaisiin monenlaisiksi monenlaisilla monenlaisille monenlaisilta monenlaisissa monenlaisista monenlaista monenlaisten monessa monesta moni monia monien moniin moniksi monilla monille monilta monissa monista monta montaa muihin muiksi muilla muille multa muissa muista muita mukana mukanaan mutta muu muuan muuhun muuksi muulla muulle muulta muun muussa muusta muut muuta muutaista muutama muutamaa muutamaan muutamaksi muutamalla muutamalle muutamalta muutaman muutamassa muutamasta muutamat muutamia muutamien muutamiin muutamiksi muutamilla muutamille muutamulta muutamissa muutamista myös myöten näet näiden näihin näiksi näille nämä ne niiden niihin niiksi niille nimittäin no noiden noihiin noiksi noille nuo ohhoh ohi ohitse paikkeilla päin paitsi paljon pitkin pois poispäin sama samaa samaan samaksi samalla samalle samalta saman samassa samasta samat samoihin samoiksi samoilla samoille samoilta samoissa samoista samoja samojen se sellainen sellaiseen sellaiseksi sellaisella sellaiselle sellaiselta sellaisen sellaisessa sellaisesta sellaiset sellaisia sellaisiin sellaisiksi sellaisilla sellaisille sellaisilta sellaisissa sellaisista sellaista sellaisten sen siihen siinä siis siitä siksi sillä sille siltä silti sinä sinua sinuksi sinulla sinulle sinulta sinun sinussa sinusta sinuun sitä taakse taas tähän tai takaa takana täksi tällainen tällaiseen tällaiseksi tällaisella tällaiselle tällaiselta tällaisen tällaisessa tällaisesta tällaiset tällaisia tällaisiin tällaisiksi tällaisilla tällaisille tällaisilta tällaisissa tällaisista tällaista tällaisten tälle tämä tämän te teidän toki tosin tuo tuohon tuoksi tuolle usea useaa useaan useaksi usealla usealle usealta usean useassa useasta useat useiden useiksi useilla useille useilta useisiin useissa useista useita vähän vähemmän vähiten vai vaikka vailla varten vasten vastoin vielä vieläkin voi yli yllä ylle yltä ympäri ympärillä ympärille

French (fr.microsoft)

ces cet cette de des du es et la le les on un une

German (de.microsoft)

aber alle aller alles als am an auch auf aus bei bis dann das daß dein dem den der deren des dessen die diese dieser dieses du durch ein eine einem einen einer eines einige einigem einigen einiger einiges er es etliche etlichem etlichen etlicher etliches euer eurer für gegen habe haben hat hatte ich ihr

ihre im immer in ist jede jedem jeden jeder jedes jene jener jenes kann kein
keine keinem keinen können man manche manchem manchen mancher manches mehr mein
mit nach nicht noch nur oder schon sei sein seine seiner sich sie sind so
soll über um und unser unter vom von vor war welche welcher welches wenn
werden wessen wie wieder wir wird worden wurde zu zum zur zwei zwischen a ä b
c d e f g h i j k l m n o ö p q r s t u ü v w x y z ß A Ä B C D E F G H
I J K L M N O Ö P Q R S T U Ü V W X Y Z é

Greek (el.microsoft)

ο η το στο οι τα του εμάς εσάς εσένα εμένα σου αι ημών μένα σένα ων όντας εμέσας μας σεις τοις τω υμάς υμείς υμών εσέ μείς μού σου τού τής τόν τήν μου τόμας σάς τούς τά δικό δικός δικές δικών δική δικής δικήν δικιά δικιάν δικά δικιάς δικιές δικοί δικού δικούς δικόν της των τον την το τους τις τα τη ένας μια ένα ενός μιας με σε αν εάν να δια εκ εξ επί προ υπέρ από προς και ούτε μήτε ουδέ μηδέ ή είτε μα αλλά παρά όμως ωστόσο ενώ μολονότι μόνο πώς που πριν οτί λοιπόν ώστε άρα επομένως όταν σαν καθώς αφού αφότου πριν μόλις άμα προτού ως ώσπου όσο ωστόυ όποτε κάθε γιατί επειδή ίσως παρά θα ας τι αντί μετά κατά από προς αλλά για τες κι σ' απ' γι' συ μ' κατ' ουτ' στ' παρ' τόσο τι όσο ό,τι θα όπου δε εάν εγώ εσύ αυτός αυτή αυτό εμείς εσείς αυτοί αυτές αυτά αυτών αυτούς αυτές πιο εδώ εκεί έτσι στα στων πλέον ακόμα τώρα τότε όταν ούτως άλλως αλλιώς συνεπώς εξής τούδε εφεξής όθεν οσοδήποτε εντούτοις μολαταύτα έστω παρόλο πια καθόλου καν χωρίς οποτεδήποτε πράγματι όντως άραγε μολοντούτο απολύτως παρομοίως σάμπως άκρως υπό ειδεμή δηλαδή ήτοι μέσω περί περίπου α Α β Β γ Γ δ Δ ε Ε ζ Ζ η Η θ Θ ι Ι κ Κ λ Λ μ Μ ν Ν ξ Ξ ο Ο π Π ρ Ρ σ Σ τ Τ υ Υ φ Φ χ Χ ψ Ψ ω Ω ī ū Ī ū Ū Ÿ

Gujarati (gu.microsoft)

માં તે એ આ છે અને નો ની નું

Hebrew (he.microsoft)

של אל את

Hindi (hi.microsoft)

है हैं में और का की के वह यह

Hungarian (hu.microsoft)

a az és hogy nem is de szerint már csak meg még ez volt mint azt vagy pedig aztán ha akkor izé szintén ki után kell majd van aki azonban lesz mert illetve amely akkor lehet nagyon miatt ami sem a á b c cs d dz dzs e é f g gy h i í j k l ly m n ny o ó öő p q r s sz t ty u ú ü ū v w x y z zs A Á B C Cs D Dz Dzs E É F G Gy H I Í J K L Ly M N Ny O Ó Ö Ö P Q R S Sz T Ty U Ú Ü Ú V W X Y Z Zs

Icelandic (is.microsoft)

að í á það eru er og

Indonesian (Bahasa) (id.microsoft)

ah di dong ialah ini itu juga ke sih

Italian (it.microsoft)

a agli ai al all' alla alle allo d' degli dei del dell' della delle dello di e è gli i il in l' la le lo negli nei nel nell' nella nelle nello un' un una uno

Japanese (ja.microsoft)

a and in is it of the to の を に は が と で

Kannada (kn.microsoft)

ಅ ಈ ಮತ್ತು ಮತ್ತೆ ಇರುತ್ತಾನೆ ಇರುತ್ತಾಳೆ ಇರುತ್ತದೆ ಇದು ಇದನ್ನು ಇದರ ಅದನ್ನು ಅದರ

Latvian (lv.microsoft)

no un uz ir

Lithuanian (lt.microsoft)

ir arba yra jis ji

Malayalam (ml.microsoft)

ഒരു അതിന് അതിൻ്റെ ഇത് അത് ഇന്ന് അ

Malay (Latin) (ms.microsoft)

adalah atau dalam dan di ia ialah ini itu juga lah

Marathi (mr.microsoft)

होते ला होता होती होतो होतं हा हे ही ह्या हें हीं ती ते तीं ला तें तीं लां

Norwegian (nb.microsoft)

av og en ei et til i er den det

Polish (pl.microsoft)

a do i jest na o ta te to w we z za się że od przez po dla jak tym ale ma co czy oraz może tego tylko jednak jego już lub ich ze tak być być jestem jesteś jest jesteśmy jesteście są będą będziesz będzie będziemy będziecie będą byłem byłam byłeś byłaś był była było byliśmy byłyśmy byliście byłyście byli były byżbym byłabym byżbyś byłabyś byżby byłaby byżoby bylibyśmy byłybyśmy bylibyście byłybyście byliby byłyby bądź bądźmy bądźcie będąc byżże byłaże byłoże byliże byłyże które która też także który tej przed można jej przy ten pod jeszcze gdy jako by bardzo bo jeśli tych więc bez również nawet temu tm tymi ja mnie mię mi mną ty ciebie cię tobie ci tobą on go mu jemu niego nim niemu my nas nam nami wy was wam wami oni im nich nimi mój mojego mego mojemu memu moim mym moja mojej mej moją mą moi moje moich mych me moimi mymi twój twojego twego twojemu twemu twoim twym twoja twa twojej twej twoją twą twoi twoje twe twoich twych twoimi twymi nasz naszego naszemu naszym nasza naszej naszą nasi nasze naszych naszymi wasz waszego waszemu waszym wasza waszej waszą wasi wasze waszych waszymi one a b c Ą d e Ę f g h j k l Ą m n Ą ó p q r s

ſ t u v x y ź ž A Á B Č D Ě E Ě F G H I J K L Ł M N Ñ Ó Ó P Q R S Ś T U
V W X Y Z Ž

Portuguese (Brazil) (pt-Br.microsoft)

à às é a ao aos as da das de do dos e em na nas no nos o os para um uma
umas uns

Portuguese (Portugal) (pt-Pt.microsoft)

à às é a ao aos as da das de do dos e em na nas no nos o os para um uma
umas uns

Punjabi (pa.microsoft)

ਹੈ ਹਨ ਦਾ ਦੇ ਦੀ ਦੀਆਂ ਵਿਚ ਅਤੇ ਇਹ ਉਹ

Romanian (ro.microsoft)

a ai al ale alor de din este în intr la o pe și un unei unor unui

Russian (ru.microsoft)

во бы не на что по для как от это из за только или их все его он но до же то
так уже а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч щ ъ ѿ є ю
я

Serbian (Cyrillic) (sr-cyrillic.microsoft)

и је ј да се на су за од са а из о

Serbian (Latin) (sr-latin.microsoft)

i je u se su a

Slovak (sk.microsoft)

z od na k o na a k v vo na je ono

Slovenian (sl.microsoft)

in k h v je ono onega onemu onem onim

Spanish (es.microsoft)

a al de del e el en es la las lo los un una unas unos y

Swedish (sv.microsoft)

av och en ett till i är den det

Tamil (ta.microsoft)

அது இது ஒரு அல்லது இந்த அந்த

Telugu (te.microsoft)

అతను ఆయన వారు వాండుళు వాళుళు ఆమె ఆనిడ అది అవి ఇతను ఈయవ వీరు వీండుళు
ఈమె ఈనిడ

Thai (th.microsoft)

ນະ ຄວັບ ຄະ ລະ ສະ ພີ ຈະ ວະ ນັ້ນ ສີ ເລວ ພີ ພື ນະຈະ ນະຄະ ເຄວະ ເຄວະນະ ເຄວະນາ ທຣອກ ອົງ
ຂະ ໃຫນ ໃහນ ໃහນ໌ ໄນ ແກລະ ຈົກກາ ຕິ່ງໆ ແຮວະ ບຽມ ອື່ຍ ແກນະ ເຊຍ ໂວຍ ໂອຍ ເຈີ່ຍກ ທຣອກ ບັນ ແນະ
ໂວ ແຊ ແເນ ເວັ ແຊ ແສະ

Turkish (tr.microsoft)

ama ancak bazı bir çok da daha de değil diye en gibi göre hem her için ile
ise kadar ki sadece üzere ve veya ya a b c ç d e f g ğ h i i j k l m n o ö
پ r s ş t u ü v y z A B C Ç D E F G Ğ H I İ J K L M N O Ö P R S Ş T U Ü
V Y Z ben sen o biz siz onlar bu şu hangi kendi bazı çok kim beni bana bende
benden benim benimle bensiz seni sana sende senden seninle sensiz onu ona onda
ondan onunla onsuz bizi bize bizde bizden bizimle bizsiz sizi size sizde

sizden sizinle sızsız onları onlara onlarda onlardan onların onlarla onlarsız
bazısı bazısını bazısına bazısında bazısından bazısının bazısıyla bazılarımız
bazılarınız bazılarına bazılarını bazılarından bazılarıyla bazılarımızi
bazılarınızı bazılarımıza bazılarınıza bazılarımızda bazılarınızda bazılarımızdan
bazılarınızdan bazılarımıza bazılarınıza kimileri kimilerini kimilerine
kimilerinde kimilerinden kimileriyle kimilerimiz kimilerimizi kimilerimize
kimilerimizde kimilerimizden kimilerimizin kimilerimizle kimilerimzsiz
kimileriniz kimilerinizi kimilerinize kimilerinizde kimilerinizden kimilerinizin
kimilerinizle kimilerinzsiz kendim kendimi kendime kendimde kendimden kendimin
kendimle kendimsiz kendin kendini kendine kendinde kendinden kendinin kendinle
kendinsiz kendisi kendisini kendisine kendisinde kendisinden kendisiyle
kendileri kendilerini kendilerine kendilerinde kendilerinden kendileriyle
kaçımız kaçımızı kaçımıza kaçımızda kaçımızdan kaçımızla hangımız hangimizi
hangimize hangımızde hangımızden hangımızle hangimzsiz bunu buna bunda bundan
bunun bununla bunsuz bunlar bunları bunlara bunlarda bunlardan bunların
bunlarla bunlarsız şunu şuna şunda şundan şunun şununla şunsuz şunlar şunları
şunlara şunlarda şunlardan şunların şunlarla şunlarsız

Ukrainian (uk.microsoft)

і й до в у є з із

Urdu (ur.microsoft)

ان اس وہ کے کی اور کا کی بے بیں میں اے

See also

- [Tutorial: Create a custom analyzer for phone numbers](#)
- [Add language analyzers to string fields](#)
- [Add custom analyzers to string fields](#)
- [Full text search in Azure AI Search](#)
- [Analyzers for text processing in Azure AI Search](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Simple query syntax in Azure AI Search

Article • 12/10/2024

For full text search scenarios, Azure AI Search implements two Lucene-based query languages, each one aligned to a query parser. The [Simple Query Parser](#) is the default. It covers common use cases and attempts to interpret a request even if it's not perfectly composed. The other parser is [Lucene Query Parser](#) and it supports more advanced query constructions.

This article is the query syntax reference for the simple query parser.

Query syntax for both parsers applies to query expressions passed in the `search` parameter of a [query request](#), not to be confused with the [OData syntax](#), with its own syntax and rules for `filter` and `orderby` expressions in the same request.

Although the simple parser is based on the [Apache Lucene Simple Query Parser](#) class, its implementation in Azure AI Search excludes fuzzy search. If you need [fuzzy search](#), consider the alternative [full Lucene query syntax](#) instead.

Example (simple syntax)

This example shows a simple query, distinguished by `"queryType": "simple"` and valid syntax. Although query type is set below, it's the default and can be omitted unless you're reverting from an alternative type. The following example is a search over independent terms, with a requirement that all matching documents include "pool".

HTTP

```
POST https://{{service-name}}.search.windows.net/indexes/hotel-rooms-sample/docs/search?api-version=2024-07-01
{
  "queryType": "simple",
  "search": "budget hotel +pool",
  "searchMode": "all"
}
```

The `searchMode` parameter is relevant in this example. Whenever boolean operators are on the query, you should generally set `searchMode=all` to ensure that *all* of the criteria are matched. Otherwise, you can use the default `searchMode=any` that favors recall over precision.

For more examples, see [Simple query syntax examples](#). For details about the query request and parameters, see [Search Documents \(REST API\)](#).

Keyword search on terms and phrases

Strings passed to the `search` parameter can include terms or phrases in any supported language, boolean operators, precedence operators, wildcard or prefix characters for "starts with" queries, escape characters, and URL encoding characters. The `search` parameter is optional. Unspecified, `search` (`search=*` or `search=" "`) returns the top 50 documents in arbitrary (unranked) order.

- A *term search* is a query of one or more terms, where any of the terms are considered a match.
- A *phrase search* is an exact phrase enclosed in quotation marks `" "`. For example, while `Roach Motel` (without quotes) would search for documents containing `Roach` and/or `Motel` anywhere in any order, `"Roach Motel"` (with quotes) will only match documents that contain that whole phrase together and in that order (lexical analysis still applies).

Depending on your search client, you might need to escape the quotation marks in a phrase search. For example, in a POST request, a phrase search on `"Roach Motel"` in the request body might be specified as `"\"Roach Motel\""`. If you're using the Azure SDKs, the search client escapes the quotation marks when it serializes the search text. Your search phrase can be sent be as "Roach Motel".

By default, all strings passed in the `search` parameter undergo lexical analysis. Make sure you understand the tokenization behavior of the analyzer you're using. Often, when query results are unexpected, the reason can be traced to how terms are tokenized at query time. You can [test tokenization on specific strings](#) to confirm the output.

Any text input with one or more terms is considered a valid starting point for query execution. Azure AI Search will match documents containing any or all of the terms, including any variations found during analysis of the text.

As straightforward as this sounds, there's one aspect of query execution in Azure AI Search that *might* produce unexpected results, increasing rather than decreasing search results as more terms and operators are added to the input string. Whether this expansion actually occurs depends on the inclusion of a NOT operator, combined with a `searchMode` parameter setting that determines how NOT is interpreted in terms of `AND` or `OR` behaviors. For more information, see the `NOT` operator under [Boolean operators](#).

Boolean operators

You can embed Boolean operators in a query string to improve the precision of a match. In the simple syntax, boolean operators are character-based. Text operators, such as the word AND, aren't supported.

[Expand table

Character	Example	Usage
+	pool + ocean	An <code>AND</code> operation. For example, <code>pool + ocean</code> stipulates that a document must contain both terms.
	pool ocean	An <code>OR</code> operation finds a match when either term is found. In the example, the query engine will return a match on documents containing either <code>pool</code> or <code>ocean</code> or both. Because <code>OR</code> is the default conjunction operator, you could also leave it out, such that <code>pool ocean</code> is the equivalent of <code>pool ocean</code> .
-	pool - ocean	A <code>NOT</code> operation returns matches on documents that exclude the term. The <code>searchMode</code> parameter on a query request controls whether a term with the <code>NOT</code> operator is <code>ANDed</code> or <code>ORed</code> with other terms in the query (assuming there's no boolean operators on the other terms). Valid values include <code>any</code> or <code>all</code> . <code>searchMode=any</code> increases the recall of queries by including more results, and by default <code>-</code> will be interpreted as "OR NOT". For example, <code>pool - ocean</code> will match documents that either contain the term <code>pool</code> or those that don't contain the term <code>ocean</code> . <code>searchMode=all</code> increases the precision of queries by including fewer results, and by default <code>-</code> will be interpreted as "AND NOT". For example, with <code>searchMode=any</code> , the query <code>pool - ocean</code> will match documents that contain the term "pool" and all documents that don't contain the term "ocean". This is arguably a more intuitive behavior for the <code>-</code> operator. Therefore, you should consider using <code>searchMode=all</code> instead of <code>searchMode=any</code> if you want to optimize searches for precision instead of recall, and Your users frequently use the <code>-</code> operator in searches. When deciding on a <code>searchMode</code> setting, consider the user interaction patterns for queries in various applications. Users who are searching for information are more likely to include an operator in a query, as opposed to e-commerce sites that have more built-in navigation structures.

Prefix queries

For "starts with" queries, add a suffix operator (*) as the placeholder for the remainder of a term. A prefix query must begin with at least one plain text character before you can add the suffix operator.

[Expand table](#)

Character	Example	Usage
*	<code>lingui*</code> will match on "linguistic" or "linguini"	The asterisk (*) represents one or more characters of arbitrary length, ignoring case.

Similar to filters, a prefix query looks for an exact match. As such, there's no relevance scoring (all results receive a search score of 1.0). Be aware that prefix queries can be slow, especially if the index is large and the prefix consists of a small number of characters. An alternative methodology, such as edge n-gram tokenization, might perform faster. Terms using prefix search can't be longer than 1000 characters.

Simple syntax supports prefix matching only. For suffix or infix matching against the end or middle of a term, use the [full Lucene syntax for wildcard search](#).

Escaping search operators

In the simple syntax, search operators include these characters: + | " () ' \

If any of these characters are part of a token in the index, escape it by prefixing it with a single backslash (\) in the query. For example, suppose you used a custom analyzer for whole term tokenization, and your index contains the string "Luxury+Hotel". To get an exact match on this token, insert an escape character: `search=luxury\+hotel`.

To make things simple for the more typical cases, there are two exceptions to this rule where escaping isn't needed:

- The NOT operator - only needs to be escaped if it's the first character after a whitespace. If the - appears in the middle (for example, in `3352CDD0-EF30-4A2E-A512-3B30AF40F3FD`), you can skip escaping.
- The suffix operator * only needs to be escaped if it's the last character before a whitespace. If the * appears in the middle (for example, in `4*4=16`), no escaping is needed.

① Note

By default, the standard analyzer will delete and break words on hyphens, whitespace, ampersands, and other characters during [lexical analysis](#). If you require special characters to remain in the query string, you might need an analyzer that preserves them in the index. Some choices include Microsoft natural [language analyzers](#), which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see [Partial terms, patterns, and special characters](#).

Encoding unsafe and reserved characters in URLs

Ensure all unsafe and reserved characters are encoded in a URL. For example, '#' is an unsafe character because it's a fragment/anchor identifier in a URL. The character must be encoded to `%23` if used in a URL. '&' and '=' are examples of reserved characters as they delimit parameters and specify values in Azure AI Search. For more information, see [RFC1738: Uniform Resource Locators \(URL\)](#).

Unsafe characters are `" ` < > # % { } | \ ^ ~ []`. Reserved characters are `; / ? : @ = + &`.

Special characters

Special characters can range from currency symbols like '\$' or '€', to emojis. Many analyzers, including the default standard analyzer, will exclude special characters during indexing, which means they won't be represented in your index.

If you need special character representation, you can assign an analyzer that preserves them:

- The whitespace analyzer considers any character sequence separated by white spaces as tokens (so the '♥' emoji would be considered a token).
- A [language analyzer](#), such as the Microsoft English analyzer (`en.microsoft`), would take the '\$' or '€' string as a token.

For confirmation, you can [test an analyzer](#) to see what tokens are generated for a given string. As you might expect, you might not get full tokenization from a single analyzer. A workaround is to create multiple fields that contain the same content, but with different analyzer assignments (for example, `description_en`, `description_fr`, and so forth for language analyzers).

When using Unicode characters, make sure symbols are properly escaped in the query url (for instance for '♥' would use the escape sequence `%E2%9D%A4+`). Some web clients do this translation automatically.

Precedence (grouping)

You can use parentheses to create subqueries, including operators within the parenthetical statement. For example, `motel+(wifi|luxury)` will search for documents containing the "motel" term and either "wifi" or "luxury" (or both).

Query size limits

If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

- For GET, the length of the URL can't exceed 8 KB.
- For POST (and any other request), where the body of the request includes `search` and other parameters such as `filter` and `orderby`, the maximum size is 16 MB.
Additional limits include:
 - The maximum length of the search clause is 100,000 characters.
 - The maximum number of clauses in `search` (expressions separated by AND or OR) is 1024.
 - The maximum search term size is 1000 characters for [prefix search](#).
 - There's also a limit of approximately 32 KB on the size of any individual term in a query.

For more information on query limits, see [API request limits](#).

Next steps

If you'll be constructing queries programmatically, review [Full text search in Azure AI Search](#) to understand the stages of query processing and the implications of text analysis.

You can also review the following articles to learn more about query construction:

- [Query examples for simple search](#)
- [Query examples for full Lucene search](#)
- [Search Documents REST API](#)
- [Lucene query syntax](#)

- Filter and Select (OData) expression syntax
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Lucene query syntax in Azure AI Search

Article • 12/11/2024

When creating queries in Azure AI Search, you can opt for the full [Lucene Query Parser](#) syntax for specialized query forms: wildcard, fuzzy search, proximity search, regular expressions. Much of the Lucene Query Parser syntax is [implemented intact in Azure AI Search](#), except for *range searches*, which are constructed through `$filter` expressions.

To use full Lucene syntax, set the `queryType` to `full` and pass in a query expression patterned for wildcard, fuzzy search, or one of the other query forms supported by the full syntax. In REST, query expressions are provided in the `search` parameter of a [Search Documents \(REST API\)](#) request.

Example (full syntax)

The following example is a search request constructed using the full syntax. This particular example shows in-field search and term boosting. It looks for hotels where the category field contains the term `budget`. Any documents containing the phrase `"recently renovated"` are ranked higher as a result of the term boost value (3).

```
HTTP  
  
POST /indexes/hotels-sample-index/docs/search?api-version=2024-07-01  
{  
  "queryType": "full",  
  "search": "category:budget AND \"recently renovated\"^3",  
  "searchMode": "all"  
}
```

While not specific to any query type, the `searchMode` parameter is relevant in this example. Whenever operators are on the query, you should generally set `searchMode=all` to ensure that *all* of the criteria are matched.

For more examples, see [Lucene query syntax examples](#). For details about the query request and parameters, including `searchMode`, see [Search Documents \(REST API\)](#).

Syntax fundamentals

The following syntax fundamentals apply to all queries that use the Lucene syntax.

Operator evaluation in context

Placement determines whether a symbol is interpreted as an operator or just another character in a string.

For example, in Lucene full syntax, the tilde (~) is used for both fuzzy search and proximity search. When placed after a quoted phrase, ~ invokes proximity search. When placed at the end of a term, ~ invokes fuzzy search.

Within a term, such as `business~analyst`, the character isn't evaluated as an operator. In this case, assuming the query is a term or phrase query, [full text search](#) with [lexical analysis](#) strips out the ~ and breaks the term `business~analyst` in two: `business` OR `analyst`.

The example above is the tilde (~), but the same principle applies to every operator.

Escaping special characters

In order to use any of the search operators as part of the search text, escape the character by prefixing it with a single backslash (\). For example, for a wildcard search on `https://`, where `://` is part of the query string, you would specify `search=https\:**`. Similarly, an escaped phone number pattern might look like this `\+1 \(800\)-642\(-7676`.

Special characters that require escaping include the following:

+ - & | ! () { } [] ^ " ~ * ? : \ /

ⓘ Note

Although escaping keeps tokens together, [lexical analysis](#) during indexing may strip them out. For example, the standard Lucene analyzer will break words on hyphens, whitespace, and other characters. If you require special characters in the query string, you might need an analyzer that preserves them in the index. Some choices include Microsoft natural [language analyzers](#), which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see [Partial terms, patterns, and special characters](#).

Encoding unsafe and reserved characters in URLs

Ensure all unsafe and reserved characters are encoded in a URL. For example, # is an unsafe character because it's a fragment/anchor identifier in a URL. The character must be encoded to %23 if used in a URL. & and = are examples of reserved characters as they delimit parameters and specify values in Azure AI Search. See [RFC1738: Uniform Resource Locators \(URL\)](#) for more details.

Unsafe characters are " ` < > # % { } | \ ^ ~ []. Reserved characters are ; / ? : @ = + &.

Boolean operators

You can embed Boolean operators in a query string to improve the precision of a match. The full syntax supports text operators in addition to character operators. Always specify text boolean operators (AND, OR, NOT) in all caps.

[Expand table](#)

Text operator	Character operator	Example	Usage
AND	+	wifi AND luxury	Specifies terms that a match must contain. In the example, the query engine looks for documents containing both wifi and luxury. The plus character (+) can also be used directly in front of a term to make it required. For example, +wifi +luxury stipulates that both terms must appear somewhere in the field of a single document.
OR	(none) ¹	wifi OR luxury	Finds a match when either term is found. In the example, the query engine returns match on documents containing either wifi or luxury or both. Because OR is the default conjunction operator, you could also leave it out, such that wifi luxury is the equivalent of wifi OR luxury.
NOT	!, -	wifi - luxury	Returns a match on documents that exclude the term. For example, wifi -luxury searches for documents that have the wifi term but not luxury.

¹ The | character isn't supported for OR operations.

NOT Boolean operator

 **Important**

The NOT operator (`NOT`, `!`, or `-`) behaves differently in full syntax than it does in simple syntax.

- In simple syntax, queries with negation always have a wildcard automatically added. For example, the query `-luxury` is automatically expanded to `-luxury *`.
- In full syntax, queries with negation cannot be combined with a wildcard. For example, the queries `-luxury *` is not allowed.
- In full syntax, queries with a single negation are not allowed. For example, the query `-luxury` is not allowed.
- In full syntax, negations will behave as if they are always ANDed onto the query regardless of the search mode.
 - For example, the full syntax query `wifi -luxury` in full syntax only fetches documents that contain the term `wifi`, and then applies the negation `-luxury` to those documents.
- If you want to use negations to search over all documents in the index, simple syntax with the `any` search mode is recommended.
- If you want to use negations to search over a subset of documents in the index, full syntax or the simple syntax with the `all` search mode are recommended.

[\[+\] Expand table](#)

Query Type	Search Mode	Example Query	Behavior
Simple	any	<code>wifi -luxury</code>	Returns all documents in the index. Documents with the term "wifi" or documents missing the term "luxury" are ranked higher than other documents. The query is expanded to <code>wifi OR -luxury OR *</code> .
Simple	all	<code>wifi -luxury</code>	Returns only documents in the index that contain the term "wifi" and don't contain the term "luxury". The query is expanded to <code>wifi AND -luxury AND *</code> .
Full	any	<code>wifi -luxury</code>	Returns only documents in the index that contain the term "wifi", and then documents that contain the term "luxury" are removed from the results.
Full	all	<code>wifi -luxury</code>	Returns only documents in the index that contain the term "wifi", and then documents that contain the term "luxury" are removed from the results.

Fielded search

You can define a fielded search operation with the `fieldName:searchExpression` syntax, where the search expression can be a single word or a phrase, or a more complex expression in parentheses, optionally with Boolean operators. Some examples include the following:

- `genre:jazz NOT history`
- `artists:(“Miles Davis” “John Coltrane”)`

Be sure to put multiple strings within quotation marks if you want both strings to be evaluated as a single entity, in this case searching for two distinct artists in the `artists` field.

The field specified in `fieldName:searchExpression` must be a `searchable` field. See [Create Index](#) for details on how index attributes are used in field definitions.

ⓘ Note

When using fielded search expressions, you do not need to use the `searchFields` parameter because each fielded search expression has a field name explicitly specified. However, you can still use the `searchFields` parameter if you want to run a query where some parts are scoped to a specific field, and the rest could apply to several fields. For example, the query `search=genre:jazz NOT history&searchFields=description` would match `jazz` only to the `genre` field, while it would match `NOT history` with the `description` field. The field name provided in `fieldName:searchExpression` always takes precedence over the `searchFields` parameter, which is why in this example, we do not need to include `genre` in the `searchFields` parameter.

Fuzzy search

A fuzzy search finds matches in terms that have a similar construction, expanding a term up to the maximum of 50 terms that meet the distance criteria of two or less. For more information, see [Fuzzy search](#).

To do a fuzzy search, use the tilde `~` symbol at the end of a single word with an optional parameter, a number between 0 and 2 (default), that specifies the edit distance. For example, `blue~` or `blue~1` would return `blue`, `blues`, and `glue`.

Fuzzy search can only be applied to terms, not quotation-enclosed phrases, but you can append the tilde to each term individually in a multi-part name or phrase. For example,

Unviersty~ of~ Wshington~ would match on University of Washington.

Proximity search

Proximity searches are used to find terms that are near each other in a document. Insert a tilde ~ symbol at the end of a phrase followed by the number of words that create the proximity boundary. For example, "hotel airport"~5 finds the terms hotel and airport within five words of each other in a document.

Term boosting

Term boosting refers to ranking a document higher if it contains the boosted term, relative to documents that don't contain the term. This differs from scoring profiles in that scoring profiles boost certain fields, rather than specific terms.

The following example helps illustrate the differences. Suppose that there's a scoring profile that boosts matches in a certain field, say *genre* in the [musicstoreindex](#) example. Term boosting could be used to further boost certain search terms higher than others. For example, rock^2 electronic boosts documents that contain the search terms in the genre field higher than other searchable fields in the index. Further, documents that contain the search term *rock* are ranked higher than the other search term *electronic* as a result of the term boost value (2).

To boost a term, use the caret, ^, symbol with a boost factor (a number) at the end of the term you're searching. You can also boost phrases. The higher the boost factor, the more relevant the term is relative to other search terms. By default, the boost factor is 1. Although the boost factor must be positive, it can be less than 1 (for example, 0.20).

Regular expression search

A regular expression search finds a match based on patterns that are valid under Apache Lucene, as documented in the [RegExp class](#).

In Azure AI Search, a regular expression is:

- Enclosed between forward slashes /
- Lower-case only

For example, to find documents containing motel or hotel, specify /[mh]otel/. Regular expression searches are matched against single words.

Some tools and languages impose extra escape character requirements beyond the [escape rules](#) imposed by Azure AI Search. For JSON, strings that include a forward slash are escaped with a backward slash: `microsoft.com/azure/` becomes `search=/.*microsoft.com\azure\/.*/` where `search=/.* <string-placeholder>.*/` sets up the regular expression, and `microsoft.com\azure\/` is the string with an escaped forward slash.

Two common symbols in regex queries are `.` and `*`. A `.` matches any one character and a `*` matches the previous character zero or more times. For example, `/be./` matches the terms `bee` and `bet` while `/be*/` would match `be`, `bee`, and `beee` but not `bet`. Together, `.*` allow you to match any series of characters so `/be.*/` would match any term that starts with `be` such as `better`.

If you get syntax errors in your regular expression, review the [escape rules](#) for special characters. You might also try a different client to confirm whether the problem is tool-specific.

Wildcard search

You can use generally recognized syntax for multiple (*) or single (?) character wildcard searches. Full Lucene syntax supports prefix and infix matching. Use [regular expression](#) syntax for suffix matching.

Note the Lucene query parser supports the use of these symbols with a single term, and not a phrase.

[] Expand table

Affix type	Description and examples
prefix	Term fragment comes before <code>*</code> or <code>?</code> . For example, a query expression of <code>search=alpha*</code> returns <code>alphanumeric</code> or <code>alphabetical</code> . Prefix matching is supported in both simple and full syntax.
suffix	Term fragment comes after <code>*</code> or <code>?</code> , with a forward slash to delimit the construct. For example, <code>search=/.*numeric/</code> returns <code>alphanumeric</code> .
infix	Term fragments enclose <code>*</code> or <code>?</code> . For example, <code>search=non*al</code> returns <code>non-numerical</code> and <code>nonsensical</code> .

You can combine operators in one expression. For example, `980?2*` matches on `98072-1222` and `98052-1234`, where `?` matches on a single (required) character, and `*` matches

on characters of an arbitrary length that follow.

Suffix matching requires the regular expression forward slash / delimiters. Generally, you can't use a * or ? symbol as the first character of a term, without the /. It's also important to note that the * behaves differently when used outside of regex queries. Outside of the regex forward slash / delimiter, the * is a wildcard character and matches any series of characters much like .* in regex. As an example, `search=/non.*a1/` produces the same result set as `search=non*a1`.

ⓘ Note

As a rule, pattern matching is slow so you might want to explore alternative methods, such as edge n-gram tokenization that creates tokens for sequences of characters in a term. With n-gram tokenization, the index will be larger, but queries might execute faster, depending on the pattern construction and the length of strings you are indexing. For more information, see [Partial term search and patterns with special characters](#).

Effect of an analyzer on wildcard queries

During query parsing, queries that are formulated as prefix, suffix, wildcard, or regular expressions are passed as-is to the query tree, bypassing [lexical analysis](#). Matches will only be found if the index contains the strings in the format your query specifies. In most cases, you need an analyzer during indexing that preserves string integrity so that partial term and pattern matching succeeds. For more information, see [Partial term search in Azure AI Search queries](#).

Consider a situation where you might want the search query `terminal*` to return results that contain terms such as `terminate`, `termination`, and `terminates`.

If you were to use the en.lucene (English Lucene) analyzer, it would apply aggressive stemming of each term. For example, `terminate`, `termination`, `terminates` will all be tokenized down to the token `termi` in your index. On the other side, terms in queries using wildcards or fuzzy search aren't analyzed at all, so there would be no results that would match the `terminat*` query.

On the other side, the Microsoft analyzers (in this case, the en.microsoft analyzer) are a bit more advanced and use lemmatization instead of stemming. This means that all generated tokens should be valid English words. For example, `terminate`, `terminates`,

and `termination` will mostly stay whole in the index, and would be a preferable choice for scenarios that depend a lot on wildcards and fuzzy search.

Scoring wildcard and regex queries

Azure AI Search uses frequency-based scoring ([BM25 ↗](#)) for text queries. However, for wildcard and regex queries where scope of terms can potentially be broad, the frequency factor is ignored to prevent the ranking from biasing towards matches from rarer terms. All matches are treated equally for wildcard and regex searches.

Special characters

In some circumstances, you might want to search for a special character, like an '♥' emoji or the '€' sign. In such cases, make sure that the analyzer you use doesn't filter those characters out. The standard analyzer bypasses many special characters, excluding them from your index.

Analyzers that tokenize special characters include the whitespace analyzer, which takes into consideration any character sequences separated by whitespaces as tokens (so the ❤ string would be considered a token). Also, a language analyzer like the Microsoft English analyzer ("en.microsoft"), would take the "€" string as a token. You can [test an analyzer](#) to see what tokens it generates for a given query.

When using Unicode characters, make sure symbols are properly escaped in the query url (for instance for ❤ would use the escape sequence `%E2%9D%A4+`). Some REST clients do this translation automatically.

Precedence (grouping)

You can use parentheses to create subqueries, including operators within the parenthetical statement. For example, `motel+(wifi|luxury)` searches for documents containing the `motel` term and either `wifi` or `luxury` (or both).

Field grouping is similar but scopes the grouping to a single field. For example, `hotelAmenities:(gym+(wifi|pool))` searches the field `hotelAmenities` for `gym` and `wifi`, or `gym` and `pool`.

Query size limits

Azure AI Search imposes limits on query size and composition because unbounded queries can destabilize your search service. There are limits on query size and composition (the number of clauses). Limits also exist for the length of prefix search and for the complexity of regex search and wildcard search. If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

For more information on query limits, see [API request limits](#).

See also

- [Query examples for simple search](#)
- [Query examples for full Lucene search](#)
- [Search Documents](#)
- [OData expression syntax for filters and sorting](#)
- [Simple query syntax in Azure AI Search](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

moreLikeThis (preview) in Azure AI Search

Article • 02/20/2025

ⓘ Important

This feature is in public preview under [Supplemental Terms of Use](#). The [preview REST API](#) supports this feature.

`moreLikeThis=[key]` is a query parameter in the [Search Documents API](#) that finds documents similar to the document specified by the document key. When a search request is made with `moreLikeThis`, a query is generated with search terms extracted from the given document that describe that document best. The generated query is then used to make the search request. The `moreLikeThis` parameter can't be used with the search parameter, `search=[string]`.

By default, the contents of all top-level searchable fields are considered. If you want to specify particular fields instead, you can use the `searchFields` parameter.

The `moreLikeThis` parameter isn't supported for [complex types](#) and the presence of complex types will impact your query logic. If your index is a complex type, you must set `searchFields` to the top-level searchable fields over which `moreLikeThis` iterates. For example, if the index has a searchable `field1` of type `Edm.String`, and `field2` that's a complex type with searchable subfields, the value of `searchFields` must be set to `field1` to exclude `field2`.

Examples

All following examples use the hotels sample from [Quickstart: Create a search index in the Azure portal](#).

Simple query

The following query finds documents whose description fields are most similar to the field of the source document as specified by the `moreLikeThis` parameter:

HTTP

```
GET /indexes/hotels-sample-index/docs?  
moreLikeThis=29&searchFields=Description&api-version=2024-05-01-preview
```

In this example, the request searches for hotels similar to the one with `HotelId` 29. Rather than using HTTP GET, you can also invoke `MoreLikeThis` using HTTP POST:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2024-05-01-preview  
{  
  "moreLikeThis": "29",  
  "searchFields": "Description"  
}
```

Apply filters

`MoreLikeThis` can be combined with other common query parameters like `$filter`. For instance, the query can be restricted to only hotels whose category is 'Budget' and where the rating is higher than 3.5:

HTTP

```
GET /indexes/hotels-sample-index/docs?  
moreLikeThis=20&searchFields=Description&$filter=(Category eq 'Budget' and  
Rating gt 3.5)&api-version=2024-05-01-preview
```

Select fields and limit results

The `$top` selector can be used to limit how many results should be returned in a `MoreLikeThis` query. Also, fields can be selected with `$select`. Here the top three hotels are selected along with their ID, Name, and Rating:

HTTP

```
GET /indexes/hotels-sample-index/docs?  
moreLikeThis=20&searchFields=Description&$filter=(Category eq 'Budget' and  
Rating gt 3.5)&$top=3&$select=HotelId,HotelName,Rating&api-version=2024-05-  
01-preview
```

Next steps

You can use any REST client for this exercise.

Quickstart: Text search using REST

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData language overview for `$filter`, `$orderby`, and `$select` in Azure AI Search

Article • 12/10/2024

This article provides an overview of the OData expression language used in `$filter`, `$order-by`, and `$select` expressions for keyword search in Azure AI Search over numeric and string (nonvector) fields.

The language is presented "bottom-up" starting with the most basic elements. The OData expressions that you can construct in a query request range from simple to highly complex, but they all share common elements. Shared elements include:

- **Field paths**, which refer to specific fields of your index.
- **Constants**, which are literal values of a certain data type.

Once you understand these common concepts, you can continue with the top-level syntax for each expression:

- **`$filter`** expressions are evaluated during query parsing, constraining search to specific fields or adding match criteria used during index scans.
- **`$orderby`** expressions are applied as a post-processing step over a result set to sort the documents that are returned.
- **`$select`** expressions determine which document fields are included in the result set.

The syntax of these expressions is distinct from the [simple](#) or [full](#) query syntax used in the `search` parameter, although there's some overlap in the syntax for referencing fields.

For examples in other languages such as Python or C#, see the examples in the [azure-search-vector-samples](#) repository.

ⓘ Note

Terminology in Azure AI Search differs from the [OData standard](#) in a few ways. What we call a **field** in Azure AI Search is called a **property** in OData, and similarly for **field path** versus **property path**. An **index** containing **documents** in Azure AI Search is referred to more generally in OData as an **entity set** containing **entities**. The Azure AI Search terminology is used throughout this reference.

Field paths

The following EBNF (Extended Backus-Naur Form [↗](#)) defines the grammar of field paths.

```
OData  
field_path ::= identifier('/'identifier)*  
identifier ::= [a-zA-Z_][a-zA-Z_0-9]*
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

 **Note**

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

A field path is composed of one or more **identifiers** separated by slashes. Each identifier is a sequence of characters that must start with an ASCII letter or underscore, and contain only ASCII letters, digits, or underscores. The letters can be upper- or lowercase.

An identifier can refer either to the name of a field, or to a **range variable** in the context of a [collection expression](#) (`any` or `all`) in a filter. A range variable is like a loop variable that represents the current element of the collection. For complex collections, that variable represents an object, which is why you can use field paths to refer to subfields of the variable. This is analogous to dot notation in many programming languages.

Examples of field paths are shown in the following table:

 [Expand table](#)

Field path	Description
<code>HotelName</code>	Refers to a top-level field of the index
<code>Address/City</code>	Refers to the <code>City</code> subfield of a complex field in the index; <code>Address</code> is of type <code>Edm.ComplexType</code> in this example
<code>Rooms/Type</code>	Refers to the <code>Type</code> subfield of a complex collection field in the index; <code>Rooms</code> is of type <code>Collection(Edm.ComplexType)</code> in this example

Field path	Description
Stores/Address/Country	Refers to the <code>Country</code> subfield of the <code>Address</code> subfield of a complex collection field in the index; <code>Stores</code> is of type <code>Collection(Edm.ComplexType)</code> and <code>Address</code> is of type <code>Edm.ComplexType</code> in this example
room/Type	Refers to the <code>Type</code> subfield of the <code>room</code> range variable, for example in the filter expression <code>Rooms/any(room: room/Type eq 'deluxe')</code>
store/Address/Country	Refers to the <code>Country</code> subfield of the <code>Address</code> subfield of the <code>store</code> range variable, for example in the filter expression <code>Stores/any(store: store/Address/Country eq 'Canada')</code>

The meaning of a field path differs depending on the context. In filters, a field path refers to the value of a *single instance* of a field in the current document. In other contexts, such as `$orderby`, `$select`, or in [fielded search in the full Lucene syntax](#), a field path refers to the field itself. This difference has some consequences for how you use field paths in filters.

Consider the field path `Address/City`. In a filter, this refers to a single city for the current document, like "San Francisco". In contrast, `Rooms/Type` refers to the `Type` subfield for many rooms (like "standard" for the first room, "deluxe" for the second room, and so on). Since `Rooms/Type` doesn't refer to a *single instance* of the subfield `Type`, it can't be used directly in a filter. Instead, to filter on room type, you would use a [lambda expression](#) with a range variable, like this:

OData
<code>Rooms/any(room: room/Type eq 'deluxe')</code>

In this example, the range variable `room` appears in the `room/Type` field path. That way, `room/Type` refers to the type of the current room in the current document. This is a single instance of the `Type` subfield, so it can be used directly in the filter.

Using field paths

Field paths are used in many parameters of the [Azure AI Search REST APIs](#). The following table lists all the places where they can be used, plus any restrictions on their usage:

[Expand table](#)

API	Parameter name	Restrictions
Create or Update Index	suggesters/sourceFields	None
Create or Update Index	scoringProfiles/text/weights	Can only refer to searchable fields
Create or Update Index	scoringProfiles/functions/fieldName	Can only refer to filterable fields
Search	search when queryType is full	Can only refer to searchable fields
Search	facet	Can only refer to facetable fields
Search	highlight	Can only refer to searchable fields
Search	searchFields	Can only refer to searchable fields
Suggest and Autocomplete	searchFields	Can only refer to fields that are part of a suggester
Search, Suggest, and Autocomplete	\$filter	Can only refer to filterable fields
Search and Suggest	\$orderby	Can only refer to sortable fields
Search, Suggest, and Lookup	\$select	Can only refer to retrievable fields

Constants

Constants in OData are literal values of a given [Entity Data Model \(EDM\)](#) type. See [Supported data types](#) for a list of supported types in Azure AI Search. Constants of collection types aren't supported.

The following table shows examples of constants for each of the nonvector data types that support OData expressions:

[] [Expand table](#)

Data type	Example constants
Edm.Boolean	true, false

Data type	Example constants
Edm.DateTimeOffset	2019-05-06T12:30:05.451Z
Edm.Double	3.14159, -1.2e7, NaN, INF, -INF
Edm.GeographyPoint	geography 'POINT(-122.131577 47.678581)'
Edm.GeographyPolygon	geography 'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'
Edm.Int32	123, -456
Edm.Int64	283032927235
Edm.String	'hello'

Escaping special characters in string constants

String constants in OData are delimited by single quotes. If you need to construct a query with a string constant that might itself contain single quotes, you can escape the embedded quotes by doubling them.

For example, a phrase with an unformatted apostrophe like "Alice's car" would be represented in OData as the string constant '`Alice''s car`'.

ⓘ Important

When constructing filters programmatically, it's important to remember to escape string constants that come from user input. This is to mitigate the possibility of [injection attacks](#), especially when using filters to implement [security trimming](#).

Constants syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar for most of the constants shown in the above table. The grammar for geo-spatial types can be found in [OData geo-spatial functions in Azure AI Search](#).

OData

```
constant ::=  
    string_literal  
    | date_time_offset_literal  
    | integer_literal  
    | float_literal
```

```

| boolean_literal
| 'null'

string_literal ::= '"'([^\"] | \"")*"""

date_time_offset_literal ::= date_part'T' time_part time_zone

date_part ::= year'-'month'-'day

time_part ::= hour':'minute(':'second('.'fractional_seconds)?)?

zero_to_fifty_nine ::= [0-5]digit

digit ::= [0-9]

year ::= digit digit digit digit

month ::= '0'[1-9] | '1'[0-2]

day ::= '0'[1-9] | [1-2]digit | '3'[0-1]

hour ::= [0-1]digit | '2'[0-3]

minute ::= zero_to_fifty_nine

second ::= zero_to_fifty_nine

fractional_seconds ::= integer_literal

time_zone ::= 'Z' | sign hour':'minute

sign ::= '+' | '-'

/* In practice integer literals are limited in length to the precision of
the corresponding EDM data type. */
integer_literal ::= digit+

float_literal ::=
    sign? whole_part fractional_part? exponent?
    | 'NaN'
    | '-INF'
    | 'INF'

whole_part ::= integer_literal

fractional_part ::= '.'integer_literal

exponent ::= 'e' sign? integer_literal

boolean_literal ::= 'true' | 'false'

```

An interactive syntax diagram is also available:

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

Building expressions from field paths and constants

Field paths and constants are the most basic part of an OData expression, but they're already full expressions themselves. In fact, the `$select` parameter in Azure AI Search is nothing but a comma-separated list of field paths, and `$orderby` isn't much more complicated than `$select`. If you happen to have a field of type `Edm.Boolean` in your index, you can even write a filter that is nothing but the path of that field. The constants `true` and `false` are likewise valid filters.

However, it's more common to have complex expressions that refer to more than one field and constant. These expressions are built in different ways depending on the parameter.

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar for the `$filter`, `$orderby`, and `$select` parameters. These are built up from simpler expressions that refer to field paths and constants:

OData

```
filter_expression ::= boolean_expression  
  
order_by_expression ::= order_by_clause(' ',' order_by_clause)*  
  
select_expression ::= '*' | field_path(' ', field_path)*
```

An interactive syntax diagram is also available:

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

Next steps

The `$orderby` and `$select` parameters are both comma-separated lists of simpler expressions. The `$filter` parameter is a Boolean expression that is composed of simpler subexpressions. These subexpressions are combined using logical operators such as [and](#), [or](#), [and not](#), comparison operators such as [eq](#), [lt](#), [gt](#), and so on, and collection operators such as [any](#) and [all](#).

The `$filter`, `$orderby`, and `$select` parameters are explored in more detail in the following articles:

- [OData \\$filter syntax in Azure AI Search](#)
- [OData \\$orderby syntax in Azure AI Search](#)
- [OData \\$select syntax in Azure AI Search](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData \$filter syntax in Azure AI Search

Article • 08/28/2024

In Azure AI Search, the `$filter` parameter specifies inclusion or exclusion criteria for returning matches in search results. This article describes the OData syntax of `$filter` and provides examples.

Field path construction and constants are described in the [OData language overview in Azure AI Search](#). For more information about filter scenarios, see [Filters in Azure AI Search](#).

Syntax

A filter in the OData language is a Boolean expression, which in turn can be one of several types of expression, as shown by the following EBNF ([Extended Backus-Naur Form](#)):

```
boolean_expression ::=  
    collection_filter_expression  
    | logical_expression  
    | comparison_expression  
    | boolean_literal  
    | boolean_function_call  
    | '(' boolean_expression ')'  
    | variable  
  
/* This can be a range variable in the case of a lambda, or a field path. */  
variable ::= identifier | field_path
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

The types of Boolean expressions include:

- Collection filter expressions using `any` or `all`. These apply filter criteria to collection fields. For more information, see [OData collection operators in Azure AI Search](#).
- Logical expressions that combine other Boolean expressions using the operators `and`, `or`, and `not`. For more information, see [OData logical operators in Azure AI Search](#).
- Comparison expressions, which compare fields or range variables to constant values using the operators `eq`, `ne`, `gt`, `lt`, `ge`, and `le`. For more information, see [OData comparison operators in Azure AI Search](#). Comparison expressions are also used to compare distances between geo-spatial coordinates using the `geo.distance` function. For more information, see [OData geo-spatial functions in Azure AI Search](#).
- The Boolean literals `true` and `false`. These constants can be useful sometimes when programmatically generating filters, but otherwise don't tend to be used in practice.
- Calls to Boolean functions, including:
 - `geo.intersects`, which tests whether a given point is within a given polygon. For more information, see [OData geo-spatial functions in Azure AI Search](#).
 - `search.in`, which compares a field or range variable with each value in a list of values. For more information, see [OData search.in function in Azure AI Search](#).
 - `search.ismatch` and `search.ismatchscoring`, which execute full-text search operations in a filter context. For more information, see [OData full-text search functions in Azure AI Search](#).
- Field paths or range variables of type `Edm.Boolean`. For example, if your index has a Boolean field called `.IsEnabled` and you want to return all documents where this field is `true`, your filter expression can just be the name `.IsEnabled`.
- Boolean expressions in parentheses. Using parentheses can help to explicitly determine the order of operations in a filter. For more information on the default precedence of the OData operators, see the next section.

Operator precedence in filters

If you write a filter expression with no parentheses around its sub-expressions, Azure AI Search will evaluate it according to a set of operator precedence rules. These rules are based on which operators are used to combine sub-expressions. The following table lists groups of operators in order from highest to lowest precedence:

[\[\] Expand table](#)

Group	Operator(s)
Logical operators	not
Comparison operators	eq, ne, gt, lt, ge, le
Logical operators	and
Logical operators	or

An operator that is higher in the above table will "bind more tightly" to its operands than other operators. For example, `and` is of higher precedence than `or`, and comparison operators are of higher precedence than either of them, so the following two expressions are equivalent:

odata-filter Expr

```
Rating gt 0 and Rating lt 3 or Rating gt 7 and Rating lt 10
((Rating gt 0) and (Rating lt 3)) or ((Rating gt 7) and (Rating lt 10))
```

The `not` operator has the highest precedence of all -- even higher than the comparison operators. That's why if you try to write a filter like this:

odata-filter Expr

```
not Rating gt 5
```

You'll get this error message:

text

```
Invalid expression: A unary operator with an incompatible type was
detected. Found operand type 'Edm.Int32' for operator kind 'Not'.
```

This error happens because the operator is associated with just the `Rating` field, which is of type `Edm.Int32`, and not with the entire comparison expression. The fix is to put the operand of `not` in parentheses:

odata-filter Expr

```
not (Rating gt 5)
```

Filter size limitations

There are limits to the size and complexity of filter expressions that you can send to Azure AI Search. The limits are based roughly on the number of clauses in your filter expression. A good guideline is that if you have hundreds of clauses, you are at risk of exceeding the limit. We recommend designing your application in such a way that it doesn't generate filters of unbounded size.

💡 Tip

Using [the search.in function](#) instead of long disjunctions of equality comparisons can help avoid the filter clause limit, since a function call counts as a single clause.

Examples

Find all hotels with at least one room with a base rate less than \$200 that are rated at or above 4:

odata-filter-expr

```
$filter=Rooms/any(room: room/BaseRate lt 200.0) and Rating ge 4
```

Find all hotels other than "Sea View Motel" that have been renovated since 2010:

odata-filter-expr

```
$filter=HotelName ne 'Sea View Motel' and LastRenovationDate ge 2010-01-01T00:00:00Z
```

Find all hotels that were renovated in 2010 or later. The datetime literal includes time zone information for Pacific Standard Time:

odata-filter-expr

```
$filter=LastRenovationDate ge 2010-01-01T00:00:00-08:00
```

Find all hotels that have parking included and where all rooms are non-smoking:

odata-filter-expr

```
$filter=ParkingIncluded and Rooms/all(room: not room/SmokingAllowed)
```

- OR -

odata-filter-expr

```
$filter=ParkingIncluded eq true and Rooms/all(room: room/SmokingAllowed eq false)
```

Find all hotels that are Luxury or include parking and have a rating of 5:

odata-filter-expr

```
$filter=(Category eq 'Luxury' or ParkingIncluded eq true) and Rating eq 5
```

Find all hotels with the tag "wifi" in at least one room (where each room has tags stored in a `Collection(Edm.String)` field):

odata-filter-expr

```
$filter=Rooms/any(room: room/Tags/any(tag: tag eq 'wifi'))
```

Find all hotels with any rooms:

odata-filter-expr

```
$filter=Rooms/any()
```

Find all hotels that don't have rooms:

odata-filter-expr

```
$filter=not Rooms/any()
```

Find all hotels within 10 kilometers of a given reference point (where `Location` is a field of type `Edm.GeographyPoint`):

odata-filter-expr

```
$filter=geo.distance(Location, geography'POINT(-122.131577 47.678581)') le 10
```

Find all hotels within a given viewport described as a polygon (where `Location` is a field of type `Edm.GeographyPoint`). The polygon must be closed, meaning the first and last point sets must be the same. Also, [the points must be listed in counterclockwise order](#).

odata-filter-expr

```
$filter=geo.intersects(Location, geography'POLYGON((-122.031577  
47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577  
47.578581)))')
```

Find all hotels where the "Description" field is null. The field will be null if it was never set, or if it was explicitly set to null:

odata-filter-expr

```
$filter=Description eq null
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel'). These phrases contain spaces, and space is a default delimiter. You can specify an alternative delimiter in single quotes as the third string parameter:

odata-filter-expr

```
$filter=search.in(HotelName, 'Sea View motel,Budget hotel', ',')
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel' separated by '|':

odata-filter-expr

```
$filter=search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels where all rooms have the tag 'wifi' or 'tub':

odata-filter-expr

```
$filter=Rooms/any(room: room/Tags/any(tag: search.in(tag, 'wifi, tub')))
```

Find a match on phrases within a collection, such as 'heated towel racks' or 'hairdryer included' in tags.

odata-filter-expr

```
$filter=Rooms/any(room: room/Tags/any(tag: search.in(tag, 'heated towel  
racks,hairdryer included', ',')))
```

Find documents with the word "waterfront". This filter query is identical to a [search request](#) with `search=waterfront`.

```
odata-filter-expr
```

```
$filter=search.ismatchscoring('waterfront')
```

Find documents with the word "hostel" and rating greater or equal to 4, or documents with the word "motel" and rating equal to 5. This request couldn't be expressed without the `search.ismatchscoring` function since it combines full-text search with filter operations using `or`.

```
odata-filter-expr
```

```
$filter=search.ismatchscoring('hostel') and rating ge 4 or  
search.ismatchscoring('motel') and rating eq 5
```

Find documents without the word "luxury".

```
odata-filter-expr
```

```
$filter=not search.ismatch('luxury')
```

Find documents with the phrase "ocean view" or rating equal to 5. The `search.ismatchscoring` query will be executed only against fields `HotelName` and `Description`. Documents that matched only the second clause of the disjunction will be returned too -- hotels with `Rating` equal to 5. Those documents will be returned with score equal to zero to make it clear that they didn't match any of the scored parts of the expression.

```
odata-filter-expr
```

```
$filter=search.ismatchscoring('"ocean view"', 'Description,HotelName')  
or Rating eq 5
```

Find hotels where the terms "hotel" and "airport" are no more than five words apart in the description, and where all rooms are non-smoking. This query uses the [full Lucene query language](#).

```
odata-filter-expr
```

```
$filter=search.ismatch('"hotel airport"~5', 'Description', 'full',
```

```
'any') and not Rooms/any(room: room/SmokingAllowed)
```

Find documents that have a word that starts with the letters "lux" in the Description field. This query uses [prefix search](#) in combination with `search.ismatch`.

odata-filter-expr

```
$filter=search.ismatch('lux*', 'Description')
```

Next steps

- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData \$orderby syntax in Azure AI Search

Article • 08/28/2024

In Azure AI Search, the `$orderby` parameter specifies a custom sort order for search results. This article describes the OData syntax of `$orderby` and provides examples.

Field path construction and constants are described in the [OData language overview in Azure AI Search](#). For more information about sorting behaviors, see [Ordering results](#).

Syntax

The `$orderby` parameter accepts a comma-separated list of up to 32 **order-by clauses**. The syntax of an order-by clause is described by the following EBNF ([Extended Backus-Naur Form ↗](#)):

```
order_by_clause ::= (field_path | sortable_function) ('asc' | 'desc')?  
sortable_function ::= geo_distance_call | 'search.score()'
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

Each clause has sort criteria, optionally followed by a sort direction (`asc` for ascending or `desc` for descending). If you don't specify a direction, the default is ascending. If there are null values in the field, null values appear first if the sort is `asc` and last if the sort is `desc`.

The sort criteria can either be the path of a `sortable` field or a call to either the `geo.distance` or the `search.score` functions.

For string fields, the default [ASCII sort order](#) and default [Unicode sort order](#) will be used. By default, sorting is case sensitive but you can use a [normalizer](#) to preprocess the text before sorting to change this behavior. You can also use an `asciifolding` normalizer to convert non-ASCII characters to their ASCII equivalent, if one exists.

If multiple documents have the same sort criteria and the `search.score` function isn't used (for example, if you sort by a numeric `Rating` field and three documents all have a rating of 4), ties will be broken by document score in descending order. When document scores are the same (for example, when there's no full-text search query specified in the request), then the relative ordering of the tied documents is indeterminate.

You can specify multiple sort criteria. The order of expressions determines the final sort order. For example, to sort descending by score, followed by Rating, the syntax would be `$orderby=search.score() desc,Rating desc`.

The syntax for `geo.distance` in `$orderby` is the same as it is in `$filter`. When using `geo.distance` in `$orderby`, the field to which it applies must be of type `Edm.GeographyPoint` and it must also be `sortable`.

The syntax for `search.score` in `$orderby` is `search.score()`. The function `search.score` doesn't take any parameters.

Examples

Sort hotels ascending by base rate:

```
odata-filter-expr  
$orderby=BaseRate asc
```

Sort hotels descending by rating, then ascending by base rate (remember that ascending is the default):

```
odata-filter-expr  
$orderby=Rating desc,BaseRate
```

Sort hotels descending by rating, then ascending by distance from the given coordinates:

```
odata-filter-expr
```

```
$orderby=Rating desc,geo.distance(Location, geography'POINT(-122.131577  
47.678581)') asc
```

Sort hotels in descending order by search.score and rating, and then in ascending order by distance from the given coordinates. Between two hotels with identical relevance scores and ratings, the closest one is listed first:

odata-filter-expr

```
$orderby=search.score() desc,Rating desc,geo.distance(Location,  
geography'POINT(-122.131577 47.678581)') asc
```

See also

- [How to work with search results in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

OData \$select syntax in Azure AI Search

Article • 08/28/2024

In Azure AI Search, the **\$select** parameter specifies which fields to include in search results. This article describes the OData syntax of **\$select** and provides examples.

Field path construction and constants are described in the [OData language overview in Azure AI Search](#). For more information about search result composition, see [How to work with search results in Azure AI Search](#).

Syntax

The **\$select** parameter determines which fields for each document are returned in the query result set. The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar for the **\$select** parameter:

```
select_expression ::= '*' | field_path(',') field_path)*  
field_path ::= identifier('/'identifier)*
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

The **\$select** parameter comes in two forms:

1. A single star (*), indicating that all retrievable fields should be returned, or
2. A comma-separated list of field paths, identifying which fields should be returned.

When using the second form, you may only specify retrievable fields in the list.

If you list a complex field without specifying its subfields explicitly, all retrievable subfields will be included in the query result set. For example, assume your index has an

`Address` field with `Street`, `City`, and `Country` subfields that are all retrievable. If you specify `Address` in `$select`, the query results will include all three subfields.

Examples

Include the `HotelId`, `HotelName`, and `Rating` top-level fields in the results, and include the `City` subfield of `Address`:

odata-filter-expr

```
$select=HotelId, HotelName, Rating, Address/City
```

An example result might look like this:

JSON

```
{
  "HotelId": "1",
  "HotelName": "Stay-Kay City Hotel",
  "Rating": 4,
  "Address": {
    "City": "New York"
  }
}
```

Include the `HotelName` top-level field in the results. Include all subfields of `Address`.

Include the `Type` and `BaseRate` subfields of each object in the `Rooms` collection:

odata-filter-expr

```
$select=HotelName, Address, Rooms/Type, Rooms/BaseRate
```

An example result might look like this:

JSON

```
{
  "HotelName": "Stay-Kay City Hotel",
  "Rating": 4,
  "Address": {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY",
    "Country": "USA",
    "PostalCode": "10022"
  },
}
```

```
"Rooms": [
  {
    "Type": "Budget Room",
    "BaseRate": 9.69
  },
  {
    "Type": "Budget Room",
    "BaseRate": 8.09
  }
]
```

Next steps

- How to work with search results in Azure AI Search
- OData expression language overview for Azure AI Search
- OData expression syntax reference for Azure AI Search
- Search Documents (Azure AI Search REST API)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData collection operators in Azure AI Search - `any` and `all`

Article • 08/28/2024

When writing an [OData filter expression](#) to use with Azure AI Search, it's often useful to filter on collection fields. You can achieve this using the `any` and `all` operators.

Syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar of an OData expression that uses `any` or `all`.

```
collection_filter_expression ::=  
    field_path'/all(' lambda_expression ')'  
  | field_path'/any(' lambda_expression ')'  
  | field_path'/any()'  
  
lambda_expression ::= identifier ':' boolean_expression
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

There are three forms of expression that filter collections.

- The first two iterate over a collection field, applying a predicate given in the form of a lambda expression to each element of the collection.
 - An expression using `all` returns `true` if the predicate is true for every element of the collection.
 - An expression using `any` returns `true` if the predicate is true for at least one element of the collection.
- The third form of collection filter uses `any` without a lambda expression to test whether a collection field is empty. If the collection has any elements, it returns

`true`. If the collection is empty, it returns `false`.

A **lambda expression** in a collection filter is like the body of a loop in a programming language. It defines a variable, called the **range variable**, that holds the current element of the collection during iteration. It also defines another boolean expression that is the filter criteria to apply to the range variable for each element of the collection.

Examples

Match documents whose `tags` field contains exactly the string "wifi":

```
text  
tags/any(t: t eq 'wifi')
```

Match documents where every element of the `ratings` field falls between 3 and 5, inclusive:

```
text  
ratings/all(r: r ge 3 and r le 5)
```

Match documents where any of the geo coordinates in the `locations` field is within the given polygon:

```
text  
locations/any(loc: geo.intersects(loc, geography'POLYGON((-122.031577  
47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577  
47.578581)))')
```

Match documents where the `rooms` field is empty:

```
text  
not rooms/any()
```

Match documents where (for all rooms) the `rooms/amenities` field contains "tv", and `rooms/baseRate` is less than 100:

```
text
```

```
rooms/all(room: room/amenities/any(a: a eq 'tv') and room/baseRate lt 100.0)
```

Limitations

Not every feature of filter expressions is available inside the body of a lambda expression. The limitations differ depending on the data type of the collection field that you want to filter. The following table summarizes the limitations.

[Expand table](#)

Data type	Features allowed in lambda expressions with <code>any</code>	Features allowed in lambda expressions with <code>all</code>
<code>Collection(Edm.ComplexType)</code>	Everything except <code>search.ismatch</code> and <code>search.ismatchscoring</code>	Same
<code>Collection(Edm.String)</code>	Comparisons with <code>eq</code> or <code>search.in</code>	Comparisons with <code>ne</code> or <code>not search.in()</code>
	Combining sub-expressions with <code>or</code>	Combining sub-expressions with <code>and</code>
<code>Collection(Edm.Boolean)</code>	Comparisons with <code>eq</code> or <code>ne</code>	Same
<code>Collection(Edm.GeographyPoint)</code>	Using <code>geo.distance</code> with <code>lt</code> or <code>le</code> <code>geo.intersects</code>	Using <code>geo.distance</code> with <code>gt</code> or <code>ge</code> <code>not geo.intersects(...)</code>
	Combining sub-expressions with <code>or</code>	Combining sub-expressions with <code>and</code>
<code>Collection(Edm.DateTimeOffset)</code> , <code>Collection(Edm.Double)</code> , <code>Collection(Edm.Int32)</code> , <code>Collection(Edm.Int64)</code>	Comparisons using <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>gt</code> , <code>le</code> , or <code>ge</code>	Comparisons using <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>gt</code> , <code>le</code> , or <code>ge</code>
	Combining comparisons with other sub-expressions using <code>or</code>	Combining comparisons with other sub-expressions using <code>and</code>
	Combining comparisons except <code>ne</code> with other sub-expressions using <code>and</code>	Combining comparisons except <code>eq</code> with other sub-expressions using <code>or</code>
	Expressions using	Expressions using

Data type	Features allowed in lambda expressions with <code>any</code>	Features allowed in lambda expressions with <code>all</code>
	combinations of <code>and</code> and <code>or</code> in Disjunctive Normal Form (DNF) ↗	combinations of <code>and</code> and <code>or</code> in Conjunctive Normal Form (CNF) ↗

For more details on these limitations as well as examples, see [Troubleshooting collection filters in Azure AI Search](#). For more in-depth information on why these limitations exist, see [Understanding collection filters in Azure AI Search](#).

Next steps

- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData comparison operators in Azure AI Search - `eq`, `ne`, `gt`, `lt`, `ge`, and `le`

Article • 08/28/2024

The most basic operation in an [OData filter expression](#) in Azure AI Search is to compare a field to a given value. Two types of comparison are possible -- equality comparison, and range comparison. You can use the following operators to compare a field to a constant value:

Equality operators:

- `eq`: Test whether a field is **equal to** a constant value
- `ne`: Test whether a field is **not equal to** a constant value

Range operators:

- `gt`: Test whether a field is **greater than** a constant value
- `lt`: Test whether a field is **less than** a constant value
- `ge`: Test whether a field is **greater than or equal to** a constant value
- `le`: Test whether a field is **less than or equal to** a constant value

You can use the range operators in combination with the [logical operators](#) to test whether a field is within a certain range of values. See the [examples](#) later in this article.

ⓘ Note

If you prefer, you can put the constant value on the left side of the operator and the field name on the right side. For range operators, the meaning of the comparison is reversed. For example, if the constant value is on the left, `gt` would test whether the constant value is greater than the field. You can also use the comparison operators to compare the result of a function, such as `geo.distance`, with a value. For Boolean functions such as `search.ismatch`, comparing the result to `true` or `false` is optional.

Syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar of an OData expression that uses the comparison operators.

```
comparison_expression ::=  
    variable_or_function comparison_operator constant |  
    constant comparison_operator variable_or_function  
  
variable_or_function ::= variable | function_call  
  
comparison_operator ::= 'gt' | 'lt' | 'ge' | 'le' | 'eq' | 'ne'
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

 Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

There are two forms of comparison expressions. The only difference between them is whether the constant appears on the left- or right-hand-side of the operator. The expression on the other side of the operator must be a **variable** or a function call. A variable can be either a field name, or a range variable in the case of a [lambda expression](#).

Data types for comparisons

The data types on both sides of a comparison operator must be compatible. For example, if the left side is a field of type `Edm.DateTimeOffset`, then the right side must be a date-time constant. Numeric data types are more flexible. You can compare variables and functions of any numeric type with constants of any other numeric type, with a few limitations, as described in the following table.

 Expand table

Variable or function type	Constant value type	Limitations
<code>Edm.Double</code>	<code>Edm.Double</code>	Comparison is subject to special rules for NaN
<code>Edm.Double</code>	<code>Edm.Int64</code>	Constant is converted to <code>Edm.Double</code> , resulting in a loss of precision for values of large magnitude
<code>Edm.Double</code>	<code>Edm.Int32</code>	n/a

Variable or function type	Constant value type	Limitations
Edm.Int64	Edm.Double	Comparisons to <code>NaN</code> , <code>-INF</code> , or <code>INF</code> are not allowed
Edm.Int64	Edm.Int64	n/a
Edm.Int64	Edm.Int32	Constant is converted to <code>Edm.Int64</code> before comparison
Edm.Int32	Edm.Double	Comparisons to <code>NaN</code> , <code>-INF</code> , or <code>INF</code> are not allowed
Edm.Int32	Edm.Int64	n/a
Edm.Int32	Edm.Int32	n/a

For comparisons that are not allowed, such as comparing a field of type `Edm.Int64` to `NaN`, the Azure AI Search REST API will return an "HTTP 400: Bad Request" error.

ⓘ Important

Even though numeric type comparisons are flexible, we highly recommend writing comparisons in filters so that the constant value is of the same data type as the variable or function to which it is being compared. This is especially important when mixing floating-point and integer values, where implicit conversions that lose precision are possible.

Special cases for `null` and `NaN`

When using comparison operators, it's important to remember that all non-collection fields in Azure AI Search can potentially be `null`. The following table shows all the possible outcomes for a comparison expression where either side can be `null`:

[\[+\] Expand table](#)

Operator	Result when only the field or variable is <code>null</code>	Result when only the constant is <code>null</code>	Result when both the field or variable and the constant are <code>null</code>
<code>gt</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>lt</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error

Operator	Result when only the field or variable is <code>null</code>	Result when only the constant is <code>null</code>	Result when both the field or variable and the constant are <code>null</code>
<code>ge</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>le</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>eq</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>ne</code>	<code>true</code>	<code>true</code>	<code>false</code>

In summary, `null` is equal only to itself, and is not less or greater than any other value.

If your index has fields of type `Edm.Double` and you upload `NaN` values to those fields, you will need to account for that when writing filters. Azure AI Search implements the IEEE 754 standard for handling `NaN` values, and comparisons with such values produce non-obvious results, as shown in the following table.

[\[+\] Expand table](#)

Operator	Result when at least one operand is <code>NaN</code>
<code>gt</code>	<code>false</code>
<code>lt</code>	<code>false</code>
<code>ge</code>	<code>false</code>
<code>le</code>	<code>false</code>
<code>eq</code>	<code>false</code>
<code>ne</code>	<code>true</code>

In summary, `NaN` is not equal to any value, including itself.

Comparing geo-spatial data

You can't directly compare a field of type `Edm.GeographyPoint` with a constant value, but you can use the `geo.distance` function. This function returns a value of type `Edm.Double`, so you can compare it with a numeric constant to filter based on the distance from constant geo-spatial coordinates. See the [examples](#) below.

Comparing string data

Strings can be compared in filters for exact matches using the `eq` and `ne` operators. These comparisons are case-sensitive.

Examples

Match documents where the `Rating` field is between 3 and 5, inclusive:

```
text  
Rating ge 3 and Rating le 5
```

Match documents where the `Location` field is less than 2 kilometers from the given latitude and longitude:

```
text  
geo.distance(Location, geography'POINT(-122.031577 47.578581)') lt 2.0
```

Match documents where the `LastRenovationDate` field is greater than or equal to January 1st, 2015, midnight UTC:

```
text  
LastRenovationDate ge 2015-01-01T00:00:00.000Z
```

Match documents where the `Details/Sku` field is not `null`:

```
text  
Details/Sku ne null
```

Match documents for hotels where at least one room has type "Deluxe Room", where the string of the `Rooms/Type` field matches the filter exactly:

```
text  
Rooms/any(room: room/Type eq 'Deluxe Room')
```

Next steps

- Filters in Azure AI Search
 - OData expression language overview for Azure AI Search
 - OData expression syntax reference for Azure AI Search
 - Search Documents (Azure AI Search REST API)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData full-text search functions in Azure AI Search - `search.ismatch` and `search.ismatchscoring`

07/10/2025

Azure AI Search supports full-text search in the context of [OData filter expressions](#) via the `search.ismatch` and `search.ismatchscoring` functions. These functions allow you to combine full-text search with strict Boolean filtering in ways that aren't possible just by using the top-level `search` parameter of the [Search API](#).

! Note

The `search.ismatch` and `search.ismatchscoring` functions are only supported in filters in the [Search API](#). They aren't supported in the [Suggest](#) or [Autocomplete](#) APIs.

Syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar of the `search.ismatch` and `search.ismatchscoring` functions:

```
search_is_match_call ::=  
    'search.ismatch'('scoring')? '(' search_is_match_parameters ')'  
  
search_is_match_parameters ::=  
    string_literal(',') string_literal(',', 'query_type', 'search_mode')?  
  
query_type ::= "'full'" | "'simple'"  
  
search_mode ::= "'any'" | "'all'"
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

! Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

search.ismatch

The `search.ismatch` function evaluates a full-text search query as a part of a filter expression. Matching documents are returned in the result set. The following overloads of this function are available:

- `search.ismatch(search)`
- `search.ismatch(search, searchFields)`
- `search.ismatch(search, searchFields, queryType, searchMode)`

The parameters are defined in the following table:

[] [Expand table](#)

Parameter	Type	Description
name		
<code>search</code>	<code>Edm.String</code>	The search query (in either simple or full Lucene query syntax).
<code>searchFields</code>	<code>Edm.String</code>	Comma-separated list of searchable fields to search in; defaults to all searchable fields in the index. When you use fielded search in the <code>search</code> parameter, the field specifiers in the Lucene query override any fields specified in this parameter.
<code>queryType</code>	<code>Edm.String</code>	' <code>simple</code> ' or ' <code>full</code> '; defaults to ' <code>simple</code> '. Specifies what query language was used in the <code>search</code> parameter.
<code>searchMode</code>	<code>Edm.String</code>	' <code>any</code> ' or ' <code>all</code> ', defaults to ' <code>any</code> '. Indicates whether any or all of the search terms in the <code>search</code> parameter must be matched in order to count the document as a match. When you use the Lucene Boolean operators in the <code>search</code> parameter, they take precedence over this parameter.

All the above parameters are equivalent to the corresponding [search request parameters in the Search API](#).

The `search.ismatch` function returns a value of type `Edm.Boolean`, which allows you to compose it with other filter subexpressions using the Boolean [logical operators](#).

! Note

Azure AI Search doesn't support using `search.ismatch` or `search.ismatchscoring` inside lambda expressions. This means it isn't possible to write filters over collections of objects that can correlate full-text search matches with strict filter matches on the same object. For more information on this limitation as well as examples, see [Troubleshooting](#)

[collection filters in Azure AI Search](#). For more in-depth information on why this limitation exists, see [Understanding collection filters in Azure AI Search](#).

search.ismatchscoring

The `search.ismatchscoring` function, like the `search.ismatch` function, returns `true` for documents that match the full-text search query passed as a parameter. The difference between them is that the relevance score of documents matching the `search.ismatchscoring` query contributes to the overall document score, whereas for `search.ismatch`, the document score doesn't change. The following overloads of this function are available with parameters identical to those of `search.ismatch`:

- `search.ismatchscoring(search)`
- `search.ismatchscoring(search, searchFields)`
- `search.ismatchscoring(search, searchFields, queryType, searchMode)`

Both the `search.ismatch` and `search.ismatchscoring` functions can be used in the same filter expression.

Examples

Find documents with the word "waterfront". This filter query is identical to a [search request](#) with `search=waterfront`.

odata-filter-expr

```
search.ismatchscoring('waterfront')
```

Here's the full query syntax for this request, which you can run in Search Explorer in the Azure portal. Output consists of matches on `waterfront`, `water`, and `front`.

JSON

```
{
  "search": "*",
  "select": "HotelId, HotelName, Description",
  "searchMode": "all",
  "queryType": "simple",
  "count": true,
  "filter": "search.ismatchscoring('waterfront')"
}
```

Find documents with the word "pool" and rating greater or equal to 4, or documents with the word "motel" and equal to 3.2. Note, this request couldn't be expressed without the `search.ismatchscoring` function.

odata-filter-expr

```
search.ismatchscoring('pool') and Rating ge 4 or  
search.ismatchscoring('motel') and Rating eq 3.2
```

Here's the full query syntax for this request for Search Explorer. Output consists of matches on hotels with pools having a rating greater than 4, or motels with a rating equal to 3.2.

JSON

```
{  
  "search": "*",
  "select": "HotelId, HotelName, Description, Tags, Rating",
  "searchMode": "all",
  "queryType": "simple",
  "count": true,
  "filter": "search.ismatchscoring('pool') and Rating ge 4 or
            search.ismatchscoring('motel') and Rating eq 3.2"
}
```

Find documents without the word "luxury".

odata-filter-expr

```
not search.ismatch('luxury')
```

Here's the full query syntax for this request. Output consists of matches on the term luxury.

JSON

```
{  
  "search": "*",
  "select": "HotelId, HotelName, Description, Tags, Rating",
  "searchMode": "all",
  "queryType": "simple",
  "count": true,
  "filter": "not search.ismatch('luxury')"
}
```

Find documents with the phrase "ocean" or rating equal to 3.2. The `search.ismatchscoring` query is executed only against fields `HotelName` and `Description`.

Here's the full query syntax for this request. Documents that match only the second clause of the disjunction are returned too (specifically, hotels with `Rating` equal to `3.2`). To make it clear that those documents didn't match any of the scored parts of the expression, they're returned with score equal to zero.

JSON

```
{  
  "search": "*",  
  "select": "HotelId, HotelName, Description, Rating",  
  "searchMode": "all",  
  "queryType": "full",  
  "count": true,  
  "filter": "search.ismatchscoring('ocean', 'Description,HotelName') or Rating eq  
3.2"  
}
```

Output consists of 4 matches: hotels that mention "ocean" in the Description or Hotel Name, or hotels with a rating of 3.2. Notice the search score of zero for matches on the second clause.

JSON

```
{  
  "@odata.count": 4,  
  "value": [  
    {  
      "@search.score": 1.6076145,  
      "HotelId": "18",  
      "HotelName": "Ocean Water Resort & Spa",  
      "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views  
from every room, location near the pier, rooftop pool, waterfront dining & more.",  
      "Rating": 4.2  
    },  
    {  
      "@search.score": 1.0594962,  
      "HotelId": "41",  
      "HotelName": "Windy Ocean Motel",  
      "Description": "Oceanfront hotel overlooking the beach features rooms with a  
private balcony and 2 indoor and outdoor pools. Inspired by the natural beauty of  
the island, each room includes an original painting of local scenes by the owner.  
Rooms include a mini fridge, Keurig coffee maker, and flatscreen TV. Various shops  
and art entertainment are on the boardwalk, just steps away.",  
      "Rating": 3.5  
    },  
    {  
      "@search.score": 0,  
      "HotelId": "40",  
      "HotelName": "Trails End Motel",  
      "Description": "Only 8 miles from Downtown. On-site bar/restaurant, Free hot  
breakfast buffet, Free wireless internet, All non-smoking hotel. Only 15 miles  
from airport.",  
    }  
  ]  
}
```

```

    "Rating": 3.2
  },
  {
    "@search.score": 0,
    "HotelId": "26",
    "HotelName": "Planetary Plaza & Suites",
    "Description": "Extend Your Stay. Affordable home away from home, with
amenities like free Wi-Fi, full kitchen, and convenient laundry service.",
    "Rating": 3.2
  }
]
}

```

Find documents where the terms "hotel" and "airport" are within 5 words from each other in the description of the hotel, and where smoking isn't allowed in at least some of the rooms.

odata-filter-expr

```

search.ismatch('"hotel airport"~5', 'Description', 'full', 'any') and
Rooms/any(room: not room/SmokingAllowed)

```

Here's the full query syntax. To run in Search Explorer, escape the interior quotation marks with a backslash character.

JSON

```
{
  "search": "*",
  "select": "HotelId, HotelName, Description, Tags, Rating",
  "searchMode": "all",
  "queryType": "simple",
  "count": true,
  "filter": "search.ismatch('\"hotel airport\"~5', 'Description', 'full', 'any')
and Rooms/any(room: not room/SmokingAllowed)"
}
```

Output consists of a single document where the terms "hotel" and "airport" are within 5 words distance. Smoking is allowed for several rooms in most hotels, including the one in this search result.

JSON

```
{
  "@odata.count": 1,
  "value": [
    {
      "@search.score": 1,
      "HotelId": "40",
      "HotelName": "Trails End Motel",
      "Description": "The Trails End Motel is a cozy and comfortable place to stay. It features modern amenities like free Wi-Fi, a full kitchen, and a laundry service. The rooms are spacious and well-appointed, making for a great stay. Whether you're looking for a quiet place to relax or a base for exploring the area, this motel is a great choice.",
      "Tags": "Cozy, Comfortable, Modern Amenities, Free WiFi, Full Kitchen, Laundry Service",
      "Rating": 3.2
    }
  ]
}
```

```

        "Description": "Only 8 miles from Downtown. On-site bar/restaurant, Free hot
breakfast buffet, Free wireless internet, All non-smoking hotel. Only 15 miles
from airport.",
        "Tags": [
            "bar",
            "free wifi",
            "restaurant"
        ],
        "Rating": 3.2
    }
]
}

```

Find documents that have a word that starts with the letters "lux" in the Description field. This query uses [prefix search](#) in combination with `search.ismatch`.

odata-filter-expr

```
search.ismatch('lux*', 'Description')
```

Here's a full query:

JSON

```
{
    "search": "*",
    "select": "HotelId, HotelName, Description, Tags, Rating",
    "searchMode": "all",
    "queryType": "simple",
    "count": true,
    "filter": "search.ismatch('lux*', 'Description')"
}
```

Output consists of the following matches.

JSON

```
{
    "@odata.count": 4,
    "value": [
        {
            "@search.score": 1,
            "HotelId": "18",
            "HotelName": "Ocean Water Resort & Spa",
            "Description": "New Luxury Hotel for the vacation of a lifetime. Bay views
from every room, location near the pier, rooftop pool, waterfront dining & more.",
            "Tags": [
                "view",
                "pool",
                "restaurant"
            ]
        }
    ]
}
```

```

        ],
        "Rating": 4.2
    },
    {
        "@search.score": 1,
        "HotelId": "13",
        "HotelName": "Luxury Lion Resort",
        "Description": "Unmatched Luxury. Visit our downtown hotel to indulge in luxury accommodations. Moments from the stadium and transportation hubs, we feature the best in convenience and comfort.",
        "Tags": [
            "bar",
            "concierge",
            "restaurant"
        ],
        "Rating": 4.1
    },
    {
        "@search.score": 1,
        "HotelId": "16",
        "HotelName": "Double Sanctuary Resort",
        "Description": "5 star Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Traveler magazine. Free WiFi, Flexible check in/out, Fitness Center & espresso in room.",
        "Tags": [
            "view",
            "pool",
            "restaurant",
            "bar",
            "continental breakfast"
        ],
        "Rating": 4.2
    },
    {
        "@search.score": 1,
        "HotelId": "14",
        "HotelName": "Twin Vortex Hotel",
        "Description": "New experience in the making. Be the first to experience the luxury of the Twin Vortex. Reserve one of our newly-renovated guest rooms today.",
        "Tags": [
            "bar",
            "restaurant",
            "concierge"
        ],
        "Rating": 4.4
    }
]
}

```

Next steps

- Filters in Azure AI Search

- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

OData geo-spatial functions in Azure AI Search - `geo.distance` and `geo.intersects`

Article • 08/28/2024

Azure AI Search supports geo-spatial queries in [OData filter expressions](#) via the `geo.distance` and `geo.intersects` functions. The `geo.distance` function returns the distance in kilometers between two points, one being a field or range variable, and one being a constant passed as part of the filter. The `geo.intersects` function returns `true` if a given point is within a given polygon, where the point is a field or range variable and the polygon is specified as a constant passed as part of the filter.

The `geo.distance` function can also be used in the [\\$orderby parameter](#) to sort search results by distance from a given point. The syntax for `geo.distance` in `$orderby` is the same as it is in `$filter`. When using `geo.distance` in `$orderby`, the field to which it applies must be of type `Edm.GeographyPoint` and it must also be **sortable**.

ⓘ Note

When using `geo.distance` in the `$orderby` parameter, the field you pass to the function must contain only a single geo-point. In other words, it must be of type `Edm.GeographyPoint` and not `Collection(Edm.GeographyPoint)`. It is not possible to sort on collection fields in Azure AI Search.

Syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar of the `geo.distance` and `geo.intersects` functions, as well as the geo-spatial values on which they operate:

```
geo_distance_call ::=  
    'geo.distance(' variable ',' geo_point ')'  
    | 'geo.distance(' geo_point ',' variable ')'  
  
geo_point ::= "geography'POINT(" lon_lat ")"  
  
lon_lat ::= float_literal ' ' float_literal
```

```
geo_intersects_call ::=  
    'geo.intersects(' variable ',' geo_polygon ')'  
  
/* You need at least four points to form a polygon, where the first and  
last points are the same. */  
geo_polygon ::=  
    "geography'POLYGON((" lon_lat ',' lon_lat ',' lon_lat ',' lon_lat_list  
"))'"  
  
lon_lat_list ::= lon_lat(',' lon_lat)*
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

 **Note**

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

geo.distance

The `geo.distance` function takes two parameters of type `Edm.GeographyPoint` and returns an `Edm.Double` value that is the distance between them in kilometers. This differs from other services that support OData geo-spatial operations, which typically return distances in meters.

One of the parameters to `geo.distance` must be a geography point constant, and the other must be a field path (or a range variable in the case of a filter iterating over a field of type `Collection(Edm.GeographyPoint)`). The order of these parameters doesn't matter.

The geography point constant is of the form `geography'POINT(<longitude><latitude>)'`, where the longitude and latitude are numeric constants.

 **Note**

When using `geo.distance` in a filter, you must compare the distance returned by the function with a constant using `lt`, `le`, `gt`, or `ge`. The operators `eq` and `ne` are not supported when comparing distances. For example, this is a correct usage of `geo.distance: $filter=geo.distance(location, geography'POINT(-122.131577 47.678581)') le 5.`

geo.intersects

The `geo.intersects` function takes a variable of type `Edm.GeographyPoint` and a constant `Edm.GeographyPolygon` and returns an `Edm.Boolean` -- `true` if the point is within the bounds of the polygon, `false` otherwise.

The polygon is a two-dimensional surface stored as a sequence of points defining a bounding ring (see the [examples](#) below). The polygon needs to be closed, meaning the first and last point sets must be the same. [Points in a polygon must be in counterclockwise order](#).

Geo-spatial queries and polygons spanning the 180th meridian

For many geo-spatial query libraries formulating a query that includes the 180th meridian (near the dateline) is either off-limits or requires a workaround, such as splitting the polygon into two, one on either side of the meridian.

In Azure AI Search, geo-spatial queries that include 180-degree longitude will work as expected if the query shape is rectangular and your coordinates align to a grid layout along longitude and latitude (for example, `geo.intersects(location, geography'POLYGON((179 65, 179 66, -179 66, -179 65, 179 65))')`). Otherwise, for non-rectangular or unaligned shapes, consider the split polygon approach.

Geo-spatial functions and `null`

Like all other non-collection fields in Azure AI Search, fields of type `Edm.GeographyPoint` can contain `null` values. When Azure AI Search evaluates `geo.intersects` for a field that is `null`, the result will always be `false`. The behavior of `geo.distance` in this case depends on the context:

- In filters, `geo.distance` of a `null` field results in `null`. This means the document will not match because `null` compared to any non-null value evaluates to `false`.
- When sorting results using `$orderby`, `geo.distance` of a `null` field results in the maximum possible distance. Documents with such a field will sort lower than all others when the sort direction `asc` is used (the default), and higher than all others when the direction is `desc`.

Examples

Filter examples

Find all hotels within 10 kilometers of a given reference point (where location is a field of type `Edm.GeographyPoint`):

odata-filter Expr

```
geo.distance(location, geography'POINT(-122.131577 47.678581)') le 10
```

Find all hotels within a given viewport described as a polygon (where location is a field of type `Edm.GeographyPoint`). Note that the polygon is closed (the first and last point sets must be the same) and [the points must be listed in counterclockwise order](#).

odata-filter Expr

```
geo.intersects(location, geography'POLYGON((-122.031577 47.578581,  
-122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))')
```

Order-by examples

Sort hotels descending by `rating`, then ascending by distance from the given coordinates:

odata-filter Expr

```
rating desc,geo.distance(location, geography'POINT(-122.131577  
47.678581)') asc
```

Sort hotels in descending order by `search.score` and `rating`, and then in ascending order by distance from the given coordinates so that between two hotels with identical ratings, the closest one is listed first:

odata-filter Expr

```
search.score() desc,rating desc,geo.distance(location,  
geography'POINT(-122.131577 47.678581)') asc
```

Next steps

- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)

- OData expression syntax reference for Azure AI Search
 - Search Documents (Azure AI Search REST API)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData logical operators in Azure AI Search - `and`, `or`, `not`

Article • 08/28/2024

OData filter expressions in Azure AI Search are Boolean expressions that evaluate to `true` or `false`. You can write a complex filter by writing a series of simpler filters and composing them using the logical operators from Boolean algebra ↗:

- `and`: A binary operator that evaluates to `true` if both its left and right sub-expressions evaluate to `true`.
- `or`: A binary operator that evaluates to `true` if either one of its left or right sub-expressions evaluates to `true`.
- `not`: A unary operator that evaluates to `true` if its sub-expression evaluates to `false`, and vice-versa.

These, together with the collection operators `any` and `all`, allow you to construct filters that can express very complex search criteria.

Syntax

The following EBNF (Extended Backus-Naur Form ↗) defines the grammar of an OData expression that uses the logical operators.

```
logical_expression ::=  
    boolean_expression ('and' | 'or') boolean_expression  
    | 'not' boolean_expression
```

An interactive syntax diagram is also available:

[OData syntax diagram for Azure AI Search](#)

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

There are two forms of logical expressions: binary (`and`/`or`), where there are two sub-expressions, and unary (`not`), where there is only one. The sub-expressions can be Boolean expressions of any kind:

- Fields or range variables of type `Edm.Boolean`
- Functions that return values of type `Edm.Boolean`, such as `geo.intersects` or `search.ismatch`
- [Comparison expressions](#), such as `rating gt 4`
- [Collection expressions](#), such as `Rooms/any(room: room/Type eq 'Deluxe Room')`
- The Boolean literals `true` or `false`.
- Other logical expressions constructed using `and`, `or`, and `not`.

Important

There are some situations where not all kinds of sub-expression can be used with `and`/`or`, particularly inside lambda expressions. See [OData collection operators in Azure AI Search](#) for details.

Logical operators and `null`

Most Boolean expressions such as functions and comparisons cannot produce `null` values, and the logical operators cannot be applied to the `null` literal directly (for example, `x and null` is not allowed). However, Boolean fields can be `null`, so you need to be aware of how the `and`, `or`, and `not` operators behave in the presence of null. This is summarized in the following table, where `b` is a field of type `Edm.Boolean`:

[\[\] Expand table](#)

Expression	Result when <code>b</code> is <code>null</code>
<code>b</code>	<code>false</code>
<code>not b</code>	<code>true</code>
<code>b eq true</code>	<code>false</code>
<code>b eq false</code>	<code>false</code>
<code>b eq null</code>	<code>true</code>
<code>b ne true</code>	<code>true</code>

Expression	Result when <code>b</code> is <code>null</code>
<code>b ne false</code>	<code>true</code>
<code>b ne null</code>	<code>false</code>
<code>b and true</code>	<code>false</code>
<code>b and false</code>	<code>false</code>
<code>b or true</code>	<code>true</code>
<code>b or false</code>	<code>false</code>

When a Boolean field `b` appears by itself in a filter expression, it behaves as if it had been written `b eq true`, so if `b` is `null`, the expression evaluates to `false`. Similarly, `not b` behaves like `not (b eq true)`, so it evaluates to `true`. In this way, `null` fields behave the same as `false`. This is consistent with how they behave when combined with other expressions using `and` and `or`, as shown in the table above. Despite this, a direct comparison to `false` (`b eq false`) will still evaluate to `false`. In other words, `null` is not equal to `false`, even though it behaves like it in Boolean expressions.

Examples

Match documents where the `rating` field is between 3 and 5, inclusive:

```
odata-filter-expr
rating ge 3 and rating le 5
```

Match documents where all elements of the `ratings` field are less than 3 or greater than 5:

```
odata-filter-expr
ratings/all(r: r lt 3 or r gt 5)
```

Match documents where the `location` field is within the given polygon, and the document does not contain the term "public".

```
odata-filter-expr
geo.intersects(location, geography'POLYGON((-122.031577 47.578581,
-122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))') and
not contains(lowercase($t), 'public')
```

```
not search.ismatch('public')
```

Match documents for hotels in Vancouver, Canada where there is a deluxe room with a base rate less than 160:

odata-filter Expr

```
Address/City eq 'Vancouver' and Address/Country eq 'Canada' and  
Rooms/any(room: room/Type eq 'Deluxe Room' and room/BaseRate lt 160)
```

Next steps

- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

OData `search.in` function in Azure AI Search

Article • 08/28/2024

A common scenario in [OData filter expressions](#) is to check whether a single field in each document is equal to one of many possible values. For example, this is how some applications implement [security trimming](#) -- by checking a field containing one or more principal IDs against a list of principal IDs representing the user issuing the query. One way to write a query like this is to use the `eq` and `or` operators:

```
odata-filter-expr
```

```
group_ids/any(g: g eq '123' or g eq '456' or g eq '789')
```

However, there is a shorter way to write this, using the `search.in` function:

```
odata-filter-expr
```

```
group_ids/any(g: search.in(g, '123, 456, 789'))
```

ⓘ Important

Besides being shorter and easier to read, using `search.in` also provides [performance benefits](#) and avoids certain [size limitations of filters](#) when there are hundreds or even thousands of values to include in the filter. For this reason, we strongly recommend using `search.in` instead of a more complex disjunction of equality expressions.

ⓘ Note

Version 4.01 of the OData standard has recently introduced the [in operator](#), which has similar behavior as the `search.in` function in Azure AI Search. However, Azure AI Search does not support this operator, so you must use the `search.in` function instead.

Syntax

The following EBNF ([Extended Backus-Naur Form](#)) defines the grammar of the `search.in` function:

```
search_in_call ::=  
    'search.in(' variable ',' string_literal(',') string_literal)? ')'
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure AI Search

ⓘ Note

See [OData expression syntax reference for Azure AI Search](#) for the complete EBNF.

The `search.in` function tests whether a given string field or range variable is equal to one of a given list of values. Equality between the variable and each value in the list is determined in a case-sensitive fashion, the same way as for the `eq` operator. Therefore an expression like `search.in(myfield, 'a, b, c')` is equivalent to `myfield eq 'a' or myfield eq 'b' or myfield eq 'c'`, except that `search.in` will yield much better performance.

There are two overloads of the `search.in` function:

- `search.in(variable, valueList)`
- `search.in(variable, valueList, delimiters)`

The parameters are defined in the following table:

[+] [Expand table](#)

Parameter	Type	Description
name		
<code>variable</code>	<code>Edm.String</code>	A string field reference (or a range variable over a string collection field in the case where <code>search.in</code> is used inside an <code>any</code> or <code>all</code> expression).
<code>valueList</code>	<code>Edm.String</code>	A string containing a delimited list of values to match against the <code>variable</code> parameter. If the <code>delimiters</code> parameter is not specified, the default delimiters are space and comma.

Parameter	Type	Description
name		

`delimiters` `Edm.String` A string where each character is treated as a separator when parsing the `valueList` parameter. The default value of this parameter is `' , '`, which means that any values with spaces and/or commas between them will be separated. If you need to use separators other than spaces and commas because your values include those characters, you can specify alternate delimiters such as `'|'` in this parameter.

Performance of `search.in`

If you use `search.in`, you can expect sub-second response time when the second parameter contains a list of hundreds or thousands of values. There is no explicit limit on the number of items you can pass to `search.in`, although you are still limited by the maximum request size. However, the latency will grow as the number of values grows.

Examples

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel'. Phrases contain spaces, which is a default delimiter. You can specify an alternative delimiter in single quotes as the third string parameter:

odata-filter Expr

```
search.in(HotelName, 'Sea View motel,Budget hotel', ',')
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel' separated by '|':

odata-filter Expr

```
search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels with rooms that have the tag 'wifi' or 'tub':

odata-filter Expr

```
Rooms/any(room: room/Tags/any(tag: search.in(tag, 'wifi, tub')))
```

Find a match on phrases within a collection, such as 'heated towel racks' or 'hairdryer included' in tags.

```
odata-filter-expr
```

```
Rooms/any(room: room/Tags/any(tag: search.in(tag, 'heated towel  
racks,hairdryer included', ',')))
```

Find all hotels without the tag 'motel' or 'cabin':

```
odata-filter-expr
```

```
Tags/all(tag: not search.in(tag, 'motel, cabin'))
```

Next steps

- [Filters in Azure AI Search](#)
- [OData expression language overview for Azure AI Search](#)
- [OData expression syntax reference for Azure AI Search](#)
- [Search Documents \(Azure AI Search REST API\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData `search.score` function in Azure AI Search

Article • 08/28/2024

When you send a query to Azure AI Search without the `$orderby` parameter, the results that come back will be sorted in descending order by relevance score. Even when you do use `$orderby`, the relevance score is used to break ties by default. However, sometimes it's useful to use the relevance score as an initial sort criteria, and some other criteria as the tie-breaker. The example in this article demonstrates using the `search.score` function for sorting.

ⓘ Note

The relevance score is computed by the relevance ranking algorithm, and the range varies depending on which algorithm you use. For more information, see [Relevance and scoring in Azure AI Search](#).

Syntax

The syntax for `search.score` in `$orderby` is `search.score()`. The function `search.score` doesn't take any parameters. It can be used with the `asc` or `desc` sort-order specifier, just like any other clause in the `$orderby` parameter. It can appear anywhere in the list of sort criteria.

Example

Sort hotels in descending order by `search.score` and `rating`, and then in ascending order by distance from the given coordinates so that between two hotels with identical ratings, the closest one is listed first:

odata-filter-expr

```
search.score() desc, rating desc, geo.distance(location,  
geography'POINT(-122.131577 47.678581)') asc
```

Next steps

- OData expression language overview for Azure AI Search
 - OData expression syntax reference for Azure AI Search
 - Search Documents (Azure AI Search REST API)
-

Feedback

Was this page helpful?



Yes



No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

OData expression syntax reference for Azure AI Search

Article • 08/28/2024

Azure AI Search uses [OData expressions](#) as parameters throughout the API. Most commonly, OData expressions are used for the `$orderby` and `$filter` parameters. These expressions can be complex, containing multiple clauses, functions, and operators. However, even simple OData expressions like property paths are used in many parts of the Azure AI Search REST API. For example, path expressions are used to refer to subfields of complex fields everywhere in the API, such as when listing subfields in a [suggester](#), a [scoring function](#), the `$select` parameter, or even [fielded search in Lucene queries](#).

This article describes all these forms of OData expressions using a formal grammar. There is also an [interactive diagram](#) to help visually explore the grammar.

Formal grammar

We can describe the subset of the OData language supported by Azure AI Search using an EBNF ([Extended Backus-Naur Form](#)) grammar. Rules are listed "top-down", starting with the most complex expressions, and breaking them down into more primitive expressions. At the top are the grammar rules that correspond to specific parameters of the Azure AI Search REST API:

- `$filter`, defined by the `filter_expression` rule.
- `$orderby`, defined by the `order_by_expression` rule.
- `$select`, defined by the `select_expression` rule.
- Field paths, defined by the `field_path` rule. Field paths are used throughout the API. They can refer to either top-level fields of an index, or subfields with one or more [complex field](#) ancestors.

After the EBNF is a browsable [syntax diagram](#) that allows you to interactively explore the grammar and the relationships between its rules.

```
/* Top-level rules */

filter_expression ::= boolean_expression

order_by_expression ::= order_by_clause( ',' order_by_clause )*
```

```
select_expression ::= '*' | field_path(',') field_path)*

field_path ::= identifier('/'identifier)*

/* Shared base rules */

identifier ::= [a-zA-Z_][a-zA-Z_0-9]/*

/* Rules for $orderby */

order_by_clause ::= (field_path | sortable_function) ('asc' | 'desc')?

sortable_function ::= geo_distance_call | 'search.score()'

/* Rules for $filter */

boolean_expression ::=
    collection_filter_expression
  | logical_expression
  | comparison_expression
  | boolean_literal
  | boolean_function_call
  | '(' boolean_expression ')'
  | variable

/* This can be a range variable in the case of a lambda, or a field path. */
variable ::= identifier | field_path

collection_filter_expression ::=
    field_path'/all(' lambda_expression ')
  | field_path'/any(' lambda_expression ')
  | field_path'/any()

lambda_expression ::= identifier ':' boolean_expression

logical_expression ::=
    boolean_expression ('and' | 'or') boolean_expression
  | 'not' boolean_expression

comparison_expression ::=
    variable_or_function comparison_operator constant |
    constant comparison_operator variable_or_function

variable_or_function ::= variable | function_call

comparison_operator ::= 'gt' | 'lt' | 'ge' | 'le' | 'eq' | 'ne'

/* Rules for constants and literals */

constant ::=
```

```

string_literal
| date_time_offset_literal
| integer_literal
| float_literal
| boolean_literal
| 'null'

string_literal ::= """([^\"] | '\"'')*"""

date_time_offset_literal ::= date_part'T'time_part time_zone

date_part ::= year'-'month'-'day

time_part ::= hour':'minute(':second('.fractional_seconds)?)？

zero_to_fifty_nine ::= [0-5]digit

digit ::= [0-9]

year ::= digit digit digit digit

month ::= '0'[1-9] | '1'[0-2]

day ::= '0'[1-9] | [1-2]digit | '3'[0-1]

hour ::= [0-1]digit | '2'[0-3]

minute ::= zero_to_fifty_nine

second ::= zero_to_fifty_nine

fractional_seconds ::= integer_literal

time_zone ::= 'Z' | sign hour':'minute

sign ::= '+' | '-'

/* In practice integer literals are limited in length to the precision of
the corresponding EDM data type. */
integer_literal ::= sign? digit+

float_literal :=
    sign? whole_part fractional_part? exponent?
    | 'NaN'
    | '-INF'
    | 'INF'

whole_part ::= integer_literal

fractional_part ::= '.'integer_literal

exponent ::= 'e' sign? integer_literal

boolean_literal ::= 'true' | 'false'

```

```

/* Rules for functions */

function_call ::= 
    geo_distance_call |
    boolean_function_call

geo_distance_call ::= 
    'geo.distance(' variable ',' geo_point ')'
    | 'geo.distance(' geo_point ',' variable ')'

geo_point ::= "geography'POINT(" lon_lat ")'"

lon_lat ::= float_literal ' ' float_literal

boolean_function_call ::= 
    geo_intersects_call |
    search_in_call |
    search_is_match_call

geo_intersects_call ::= 
    'geo.intersects(' variable ',' geo_polygon ')'

/* You need at least four points to form a polygon, where the first and
last points are the same. */
geo_polygon ::= 
    "geography'POLYGON((" lon_lat ',' lon_lat ',' lon_lat ',' lon_lat_list
    ")))'"

lon_lat_list ::= lon_lat(',') lon_lat)*

search_in_call ::= 
    'search.in(' variable ',' string_literal(',') string_literal)? ')'

/* Note that it is illegal to call search.ismatch or search.ismatchscoring
from inside a lambda expression. */
search_is_match_call ::= 
    'search.ismatch('scoring')?'(' search_is_match_parameters ')'

search_is_match_parameters ::= 
    string_literal(',') string_literal(',') query_type ',' search_mode)??)?

query_type ::= "'full'" | "'simple'"

search_mode ::= "'any'" | "'all'"

```

Syntax diagram

To visually explore the OData language grammar supported by Azure AI Search, try the interactive syntax diagram:

See also

- [Filters in Azure AI Search](#)
 - [Search Documents \(Azure AI Search REST API\)](#)
 - [Lucene query syntax](#)
 - [Simple query syntax in Azure AI Search](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

az search

Manage Search.

Commands

 Expand table

Name	Description	Type	Status
az search admin-key	Manage Azure Search admin keys.	Core	GA
az search admin-key renew	Regenerates either the primary or secondary admin API key.	Core	GA
az search admin-key show	Gets the primary and secondary admin API keys for the specified Azure Cognitive Search service.	Core	GA
az search private-endpoint-connection	Manage Azure Search private endpoint connections.	Core	GA
az search private-endpoint-connection delete	Disconnects the private endpoint connection and deletes it from the search service.	Core	GA
az search private-endpoint-connection list	Gets a list of all private endpoint connections in the given service.	Core	GA
az search private-endpoint-connection show	Gets the details of the private endpoint connection to the search service in the given resource group.	Core	GA
az search private-endpoint-connection update	Update an existing private endpoint connection in a Search service in the given resource group.	Core	GA
az search private-link-resource	Manage Azure Search private link resources.	Core	GA
az search private-link-resource list	Gets a list of all supported private link resource types for the given service.	Core	GA
az search query-key	Manage Azure Search query keys.	Core	GA
az search query-key create	Generates a new query key for the specified search service.	Core	GA

Name	Description	Type	Status
az search query-key delete	Deletes the specified query key.	Core	GA
az search query-key list	Returns the list of query API keys for the given Azure Cognitive Search service.	Core	GA
az search service	Manage Service.	Core	GA
az search service admin-key	Manage Admin Key.	Core	GA
az search service admin-key list	Gets the primary and secondary admin API keys for the specified Azure AI Search service.	Core	GA
az search service admin-key regenerate	Regenerates either the primary or secondary admin API key. You can only regenerate one key at a time.	Core	GA
az search service check-name-availability	Checks whether or not the given search service name is available for use. Search service names must be globally unique since they are part of the service URI (<a href="https://<name>.search.windows.net">https://<name>.search.windows.net).	Core	GA
az search service create	Creates or updates a search service in the given resource group. If the search service already exists, all properties will be updated with the given values.	Core	GA
az search service delete	Delete a search service in the given resource group, along with its associated resources.	Core	GA
az search service list	Gets a list of all Search services in the given resource group.	Core	GA
az search service network-security-perimeter-configuration	Manage Network Security Perimeter Configuration.	Core	GA
az search service network-security-perimeter-configuration list	List a list of network security perimeter configurations for a search service.	Core	GA
az search service network-security-perimeter-configuration reconcile	Reconcile network security perimeter configuration for the Azure AI Search resource provider. This triggers a manual resync with network security perimeter configurations by ensuring the search service carries the latest configuration.	Core	GA
az search service network-security-	Get a network security perimeter configuration.	Core	GA

Name	Description	Type	Status
perimeter-configuration show			
az search service private-endpoint-connection	Manage Private Endpoint Connection.	Core	GA
az search service private-endpoint-connection delete	Delete the private endpoint connection and deletes it from the search service.	Core	GA
az search service private-endpoint-connection list	List a list of all private endpoint connections in the given service.	Core	GA
az search service private-endpoint-connection show	Get the details of the private endpoint connection to the search service in the given resource group.	Core	GA
az search service private-endpoint-connection update	Update a private endpoint connection to the search service in the given resource group.	Core	GA
az search service private-link-resource	Manage Private Link Resource.	Core	GA
az search service private-link-resource list	List a list of all supported private link resource types for the given service.	Core	GA
az search service query-key	Manage Create Query Key.	Core	GA
az search service query-key create	Create a new query key for the specified search service. You can create up to 50 query keys per service.	Core	GA
az search service query-key delete	Delete the specified query key. Unlike admin keys, query keys are not regenerated. The process for regenerating a query key is to delete and then recreate it.	Core	GA
az search service query-key list	Returns the list of query API keys for the given Azure AI Search service.	Core	GA
az search service shared-private-link-resource	Manage Shared Private Link Resource.	Core	GA
az search service shared-private-link-resource create	Create the creation or update of a shared private link resource managed by the search service in the given resource group.	Core	GA

Name	Description	Type	Status
az search service shared-private-link-resource delete	Delete the deletion of the shared private link resource from the search service.	Core	GA
az search service shared-private-link-resource list	List a list of all shared private link resources managed by the given service.	Core	GA
az search service shared-private-link-resource show	Get the details of the shared private link resource managed by the search service in the given resource group.	Core	GA
az search service shared-private-link-resource update	Update the creation or update of a shared private link resource managed by the search service in the given resource group.	Core	GA
az search service shared-private-link-resource wait	Place the CLI in a waiting state until a condition is met.	Core	GA
az search service show	Get the search service with the given name in the given resource group.	Core	GA
az search service update	Update an existing search service in the given resource group.	Core	GA
az search service upgrade	Upgrades the Azure AI Search service to the latest version available.	Core	GA
az search service wait	Place the CLI in a waiting state until a condition is met.	Core	GA
az search shared-private-link-resource	Manage Azure Search shared private link resources.	Core	GA
az search shared-private-link-resource create	Create shared privatelink resources in a Search service in the given resource group.	Core	GA
az search shared-private-link-resource delete	Initiates the deletion of the shared private link resource from the search service.	Core	GA
az search shared-private-link-resource list	Gets a list of all shared private link resources managed by the given service.	Core	GA
az search shared-private-link-resource show	Gets the details of the shared private link resource managed by the search service in the given resource group.	Core	GA

Name	Description	Type	Status
az search shared-private-link-resource update	Update shared privatelink resources in a Search service in the given resource group.	Core	GA
az search shared-private-link-resource wait	Wait for async shared private link resource operations.	Core	GA
az search usage	Manage Usage.	Core	GA
az search usage list	List a list of all Azure AI Search quota usages across the subscription.	Core	GA
az search usage show	Get the quota usage for a search SKU in the given subscription.	Core	GA

Az.Search Module

Preview

This is a Preview module. Preview modules aren't recommended for use in production environments. For more information, see <https://aka.ms/azps-refstatus>.

This topic displays help topics for the Azure AI Search Cmdlets.

Networking

[] Expand table

Cmdlet	Description
Get-AzSearchNetworkSecurityPerimeterConfiguration	Gets an Azure AI Search service network security perimeter configuration.
Invoke-AzSearchNetworkSecurityPerimeterConfigurationReconcile	Reconciles network security perimeter configuration for an Azure AI Search service.

Search

[] Expand table

Cmdlet	Description
Get-AzSearchAdminKeyPair	Gets admin key pair of the Azure AI Search service.
Get-AzSearchPrivateEndpointConnection	Gets private endpoint connection(s) to the Azure AI Search service.
Get-AzSearchPrivateLinkResource	Gets private link resource details for the Azure AI Search service.
Get-AzSearchQueryKey	Gets query key(s) of the Azure AI Search service.
Get-AzSearchService	Gets an Azure AI Search service.
Get-AzSearchSharedPrivateLinkResource	Gets shared private link resources(s) of the Azure AI Search service.
New-AzSearchAdminKey	Regenerates an admin key of the Azure AI Search service.

Cmdlet	Description
New-AzSearchQueryKey	Create a new query key for the Azure AI Search service.
New-AzSearchService	Creates an Azure AI Search service.
New-AzSearchSharedPrivateLinkResource	Creates a shared private link resource for the Azure AI Search service.
Remove-AzSearchPrivateEndpointConnection	Remove the private endpoint connection from the Azure AI Search service.
Remove-AzSearchQueryKey	Remove the query key from the Azure AI Search service.
Remove-AzSearchService	Remove an Azure AI Search service.
Remove-AzSearchSharedPrivateLinkResource	Remove the shared private link resource from the Azure AI Search service.
Set-AzSearchPrivateEndpointConnection	Update the private endpoint connection to the Azure AI Search service.
Set-AzSearchService	Update an Azure AI Search service.
Set-AzSearchSharedPrivateLinkResource	Update the shared private link resource for the Azure AI Search service.

Microsoft.Search searchServices

Bicep resource definition

The searchServices resource type can be deployed with operations that target:

For a list of changed properties in each API version, see [change log](#).

Resource format

To create a Microsoft.Search/searchServices resource, add the following Bicep to your template.

Bicep

```
resource symbolicname 'Microsoft.Search/searchServices@2025-05-01' = {
  scope: resourceSymbolicName or scope
  identity: {
    type: 'string'
    userAssignedIdentities: {
      {customized property}: {}
    }
  }
  location: 'string'
  name: 'string'
  properties: {
    authOptions: {
      aadOrApiKey: {
        aadAuthFailureMode: 'string'
      }
      apiKeyOnly: any(...)
    }
    computeType: 'string'
    dataExfiltrationProtections: [
      'string'
    ]
    disableLocalAuth: bool
    encryptionWithCmk: {
      enforcement: 'string'
    }
    endpoint: 'string'
    hostingMode: 'string'
    networkRuleSet: {
      bypass: 'string'
      ipRules: [
        {
          value: 'string'
        }
      ]
    }
    partitionCount: int
    publicNetworkAccess: 'string'
    replicaCount: int
    semanticSearch: 'string'
    upgradeAvailable: 'string'
  }
  sku: {
    name: 'string'
  }
  tags: {
    {customized property}: 'string'
  }
}
```

Property Values

Microsoft.Search/searchServices

 Expand table

Name	Description	Value
identity	The identity of the resource.	Identity
location	The geo-location where the resource lives	string (required)
name	The resource name	string (required)
properties	Properties of the search service.	SearchServiceProperties
scope	Use when creating a resource at a scope that is different than the deployment scope.	Set this property to the symbolic name of a resource to apply the extension resource .
sku	The SKU of the search service, which determines price tier and capacity limits. This property is required when creating a new search service.	Sku
tags	Resource tags	Dictionary of tag names and values. See Tags in templates

DataPlaneAadOrApiKeyAuthOption

[Expand table](#)

Name	Description	Value
aadAuthFailureMode	Describes what response the data plane API of a search service would send for requests that failed authentication.	'http401WithBearerChallenge' 'http403'

DataPlaneAuthOptions

[Expand table](#)

Name	Description	Value
aadOrApiKey	Indicates that either the API key or an access token from a Microsoft Entra ID tenant can be used for authentication.	DataPlaneAadOrApiKeyAuthOption
apiKeyOnly	Indicates that only the API key can be used for authentication.	any

EncryptionWithCmk

[Expand table](#)

Name	Description	Value
enforcement	Describes how a search service should enforce compliance if it finds objects that aren't encrypted with the customer-managed key.	'Disabled' 'Enabled' 'Unspecified'

Identity

[Expand table](#)

Name	Description
type	The type of identity used for the resource. The type 'SystemAssigned, UserAssigned' includes both an identity created by the system and a set of assigned identities. The type 'None' will remove all identities from the service.
userAssignedIdentities	The list of user identities associated with the resource. The user identity dictionary key references will be ARM resource IDs in the form: '/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identityName}'

IpRule

[Expand table](#)

Name	Description	Value
value	Value corresponding to a single IPv4 address (eg., 123.1.2.3) or an IP range in CIDR format (eg., 123.1.2.3/24) to be allowed.	string

NetworkRuleSet

[Expand table](#)

Name	Description	Value
bypass	Possible origins of inbound traffic that can bypass the rules defined in the 'ipRules' section.	'AzureServices' 'None'
ipRules	A list of IP restriction rules that defines the inbound network(s) with allowing access to the search service endpoint. At the meantime, all other public IP networks are blocked by the firewall. These restriction rules are applied only when the 'publicNetworkAccess' of the search service is 'enabled'; otherwise, traffic over public interface is not allowed even with any public IP rules, and private endpoint connections would be the exclusive access method.	IpRule[]

SearchServiceProperties

[Expand table](#)

Name	Description	Value
authOptions	Defines the options for how the data plane API of a search service authenticates requests. This cannot be set if 'disableLocalAuth' is set to true.	DataPlaneAuthOptions
computeType	Configure this property to support the search service using either the Default Compute or Azure Confidential Compute.	'Confidential' 'Default'
dataExfiltrationProtections	A list of data exfiltration scenarios that are explicitly disallowed for the search service. Currently, the only supported value is 'All' to disable all possible data export scenarios with more fine grained controls planned for the future.	String array containing any of: 'BlockAll'
disableLocalAuth	When set to true, calls to the search service will not be permitted to utilize API keys for authentication. This cannot be set to true if 'dataPlaneAuthOptions' are defined.	bool
encryptionWithCmk	Specifies any policy regarding encryption of resources (such as indexes) using customer manager keys within a search service.	EncryptionWithCmk
endpoint	The endpoint of the Azure AI Search service.	string
hostingMode	Applicable only for the standard3 SKU. You can set this property to enable up to 3 high density partitions that allow up to 1000 indexes, which is much higher than the maximum indexes allowed for any other SKU. For the standard3 SKU, the value is either 'Default' or 'HighDensity'. For all other SKUs, this value must be 'Default'.	'Default' 'HighDensity'
networkRuleSet	Network specific rules that determine how the Azure AI Search service may be reached.	NetworkRuleSet
partitionCount	The number of partitions in the search service; if specified, it can be 1, 2, 3, 4, 6, or 12. Values greater than 1 are only valid for standard SKUs. For 'standard3' services with hostingMode set to 'highDensity', the allowed values are between 1 and 3.	int Constraints: Min value = 1 Max value = 12
publicNetworkAccess	This value can be set to 'Enabled' to avoid breaking changes on existing customer resources and templates. If set to 'Disabled', traffic over public interface is not allowed, and private endpoint connections would be the exclusive access method.	'Disabled' 'Enabled' 'SecuredByPerimeter'
replicaCount	The number of replicas in the search service. If specified, it must be a value between 1 and 12 inclusive for standard SKUs or between 1 and 3 inclusive for basic SKU.	int Constraints: Min value = 1 Max value = 12
semanticSearch	Sets options that control the availability of semantic search. This configuration is only possible for certain Azure AI Search SKUs in certain locations.	'disabled' 'free' 'standard'
upgradeAvailable	Indicates if the search service has an upgrade available.	'available' 'notAvailable'

Sku

[Expand table](#)

Name	Description	Value
name	The SKU of the search service. Valid values include: 'free': Shared service. 'basic': Dedicated service with up to 3 replicas. 'standard': Dedicated service with up to 12 partitions and 12 replicas. 'standard2': Similar to standard, but with more capacity per search unit. 'standard3': The largest Standard offering with up to 12 partitions and 12 replicas (or up to 3 partitions with more indexes if you also set the hostingMode property to 'highDensity'). 'storage_optimized_I1': Supports 1TB per partition, up to 12 partitions. 'storage_optimized_I2': Supports 2TB per partition, up to 12 partitions.'	'basic' 'free' 'standard' 'standard2' 'standard3' 'storage_optimized_I1' 'storage_optimized_I2'

TrackedResourceTags

[Expand table](#)

Name	Description	Value
------	-------------	-------

UserAssignedIdentity

[Expand table](#)

Name	Description	Value
------	-------------	-------

UserAssignedManagedIdentities

[Expand table](#)

Name	Description	Value
------	-------------	-------

Usage Examples

Azure Verified Modules

The following [Azure Verified Modules](#) can be used to deploy this resource type.

[Expand table](#)

Module	Description
Search Service	AVM Resource Module for Search Service

Azure Quickstart Samples

The following [Azure Quickstart templates](#) contain Bicep samples for deploying this resource type.

[Expand table](#)

Bicep File	Description
Azure AI Foundry Network Restricted	This set of templates demonstrates how to set up Azure AI Foundry with private link and egress disabled, using Microsoft-managed keys for encryption and Microsoft-managed identity configuration for the AI resource.
Azure Cognitive Search service	This template creates an Azure Cognitive Search service
Network Secured Agent with User Managed Identity	This set of templates demonstrates how to set up Azure AI Agent Service with virtual network isolation using User Managed Identity authentication for the AI Service/AOAI connection and private network links to connect the agent to your secure data.

Bicep File	Description
Standard Agent Setup ↗	This set of templates demonstrates how to set up Azure AI Agent Service with the standard setup, meaning with managed identity authentication for project/hub connections and public internet access enabled. Agents use customer-owned, single-tenant search and storage resources. With this setup, you have full control and visibility over these resources, but you will incur costs based on your usage.

(Last updated on 10/22/2025)

Azure Policy built-in definitions for Azure Cognitive Search

Article • 12/10/2024

This page is an index of [Azure Policy](#) built-in policy definitions for Azure Cognitive Search. For additional Azure Policy built-ins for other services, see [Azure Policy built-in definitions](#).

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the [Azure Policy GitHub repo](#) ↗.

Azure Cognitive Search

 Expand table

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
[Preview]: Azure AI Search Service should be Zone Redundant ↗	Azure AI Search Service can be configured to be Zone Redundant or not. Availability zones are used when you add two or more replicas to your search service. Each replica is placed in a different availability zone within the region.	Audit, Deny, Disabled	1.0.0-preview ↗
Azure AI Services resources should have key access disabled (disable local authentication) ↗	Key access (local authentication) is recommended to be disabled for security. Azure OpenAI Studio, typically used in development/testing, requires key access and will not function if key access is disabled. After disabling, Microsoft Entra ID becomes the only access method, which allows maintaining minimum privilege principle and granular control. Learn more at: https://aka.ms/AI/auth ↗	Audit, Deny, Disabled	1.1.0 ↗
Azure AI Services resources should restrict network access ↗	By restricting network access, you can ensure that only allowed networks can access the service. This can be achieved by configuring network rules so that only applications from allowed networks can access the Azure AI service.	Audit, Deny, Disabled	3.2.0 ↗
Azure AI Services resources should use Azure Private Link ↗	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The Private Link platform reduces data leakage risks by handling the connectivity between the consumer and	Audit, Disabled	1.0.0 ↗

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
	<p>services over the Azure backbone network.</p> <p>Learn more about private links at: https://aka.ms/AzurePrivateLink/Overview</p>		
Azure Cognitive Search service should use a SKU that supports private link	<p>With supported SKUs of Azure Cognitive Search, Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your Search service, data leakage risks are reduced. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.</p>	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should disable public network access	<p>Disabling public network access improves security by ensuring that your Azure Cognitive Search service is not exposed on the public internet. Creating private endpoints can limit exposure of your Search service. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.</p>	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should have local authentication methods disabled	<p>Disabling local authentication methods improves security by ensuring that Azure Cognitive Search services exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/azure-cognitive-search/rbac. Note that while the disable local authentication parameter is still in preview, the deny effect for this policy may result in limited Azure Cognitive Search portal functionality since some features of the Portal use the GA API which does not support the parameter.</p>	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should use customer-managed keys to encrypt data at rest	<p>Enabling encryption at rest using a customer-managed key on your Azure Cognitive Search services provides additional control over the key used to encrypt data at rest. This feature is often applicable to customers with special compliance requirements to manage data encryption keys using a key vault.</p>	Audit, Deny, Disabled	1.0.0

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
Configure Azure AI Services resources to disable local key access (disable local authentication) ↗	<p>Key access (local authentication) is recommended to be disabled for security. Azure OpenAI Studio, typically used in development/testing, requires key access and will not function if key access is disabled. After disabling, Microsoft Entra ID becomes the only access method, which allows maintaining minimum privilege principle and granular control. Learn more at: https://aka.ms/AI/auth ↗</p>	DeployIfNotExists, Disabled	1.0.0 ↗
Configure Azure Cognitive Search services to disable local authentication ↗	<p>Disable local authentication methods so that your Azure Cognitive Search services exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/azure-cognitive-search/rbac ↗.</p>	Modify, Disabled	1.0.0 ↗
Configure Azure Cognitive Search services to disable public network access ↗	<p>Disable public network access for your Azure Cognitive Search service so that it is not accessible over the public internet. This can reduce data leakage risks. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints ↗.</p>	Modify, Disabled	1.0.0 ↗
Configure Azure Cognitive Search services with private endpoints ↗	<p>Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to your Azure Cognitive Search service, you can reduce data leakage risks. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints ↗.</p>	DeployIfNotExists, Disabled	1.0.0 ↗
Deploy Diagnostic Settings for Search Services to Event Hub ↗	<p>Deploys the diagnostic settings for Search Services to stream to a regional Event Hub when any Search Services which is missing this diagnostic settings is created or updated.</p>	DeployIfNotExists, Disabled	2.0.0 ↗
Deploy Diagnostic Settings for Search Services to Log Analytics workspace ↗	<p>Deploys the diagnostic settings for Search Services to stream to a regional Log Analytics workspace when any Search Services which is missing this diagnostic settings is created or updated.</p>	DeployIfNotExists, Disabled	1.0.0 ↗
Diagnostic logs in Azure AI services resources should be enabled ↗	<p>Enable logs for Azure AI services resources. This enables you to recreate activity trails for investigation purposes, when a security</p>	AuditIfNotExists, Disabled	1.0.0 ↗

Name	Description	Effect(s)	Version
(Azure portal)			(GitHub)
	incident occurs or your network is compromised		
Enable logging by category group for Search services (microsoft.search/searchservices) to Event Hub ↗	Resource logs should be enabled to track activities and events that take place on your resources and give you visibility and insights into any changes that occur. This policy deploys a diagnostic setting using a category group to route logs to an Event Hub for Search services (microsoft.search/searchservices).	DeployIfNotExists, AuditIfNotExists, Disabled	1.0.0 ↗
Enable logging by category group for Search services (microsoft.search/searchservices) to Log Analytics ↗	Resource logs should be enabled to track activities and events that take place on your resources and give you visibility and insights into any changes that occur. This policy deploys a diagnostic setting using a category group to route logs to a Log Analytics workspace for Search services (microsoft.search/searchservices).	DeployIfNotExists, AuditIfNotExists, Disabled	1.0.0 ↗
Enable logging by category group for Search services (microsoft.search/searchservices) to Storage ↗	Resource logs should be enabled to track activities and events that take place on your resources and give you visibility and insights into any changes that occur. This policy deploys a diagnostic setting using a category group to route logs to a Storage Account for Search services (microsoft.search/searchservices).	DeployIfNotExists, AuditIfNotExists, Disabled	1.0.0 ↗
Resource logs in Search services should be enabled ↗	Audit enabling of resource logs. This enables you to recreate activity trails to use for investigation purposes; when a security incident occurs or when your network is compromised	AuditIfNotExists, Disabled	5.0.0 ↗

Next steps

- See the built-ins on the [Azure Policy GitHub repo ↗](#).
- Review the [Azure Policy definition structure](#).
- Review [Understanding policy effects](#).

Feedback

Was this page helpful?

 Yes

 No

Azure AI Search monitoring data reference

07/25/2025

This article contains all the monitoring reference information for this service.

See [Monitor Azure AI Search](#) for details on the data you can collect for Azure AI Search and how to use it.

Metrics

This section lists all the automatically collected platform metrics for this service. These metrics are also part of the global list of [all platform metrics supported in Azure Monitor](#).

For information on metric retention, see [Azure Monitor Metrics overview](#).

Supported metrics for Microsoft.Search/searchServices

The following table lists the metrics available for the Microsoft.Search/searchServices resource type.

- All columns might not be present in every table.
- Some columns might be beyond the viewing area of the page. Select **Expand table** to view all available columns.

Table headings

- **Category** - The metrics group or classification.
- **Metric** - The metric display name as it appears in the Azure portal.
- **Name in REST API** - The metric name as referred to in the [REST API](#).
- **Unit** - Unit of measure.
- **Aggregation** - The default [aggregation](#) type. Valid values: Average (Avg), Minimum (Min), Maximum (Max), Total (Sum), Count.
- **Dimensions** - [Dimensions](#) available for the metric.
- **Time Grains** - [Intervals](#) at which the metric is sampled. For example, `PT1M` indicates that the metric is sampled every minute, `PT30M` every 30 minutes, `PT1H` every hour, and so on.
- **DS Export** - Whether the metric is exportable to Azure Monitor Logs via diagnostic settings. For information on exporting metrics, see [Create diagnostic settings in Azure Monitor](#).

[\[+\] Expand table](#)

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Document processed count	<code>DocumentsProcessedCount</code>	Count	Total (Sum), Count	<code>DataSourceName</code> , <code>Failed</code> , <code>IndexerName</code> , <code>IndexName</code> , <code>SkillsetName</code>	<code>PT1M</code>	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
documents processed						
Search Latency	SearchLatency	Seconds	Average	<none>	PT1M	Yes
Average search latency for the search service						
Search queries per second	SearchQueriesPerSecond	CountPerSecond	Average	<none>	PT1M	Yes
Search queries per second for the search service						
Skill execution invocation count	SkillExecutionCount	Count	Total (Sum), Count	DataSourceName, Failed, IndexerName, SkillName, SkillsetName, SkillType	PT1M	Yes
Number of skill executions						
Throttled search queries percentage	ThrottledSearchQueriesPercentage	Percent	Average	<none>	PT1M	Yes
Percentage of search queries that were throttled for the search service						

Search queries per second

This metric shows the average of the search queries per second (QPS) for the search service. It's common for queries to execute in milliseconds, so only queries that measure as seconds appear in a metric like

QPS. The minimum is the lowest value for search queries per second that was registered during that minute. Maximum is the highest value. Average is the aggregate across the entire minute.

[Expand table](#)

Aggregation type	Description
Average	The average number of seconds within a minute during which query execution occurred.
Count	The number of metrics emitted to the log within the one-minute interval.
Maximum	The highest number of search queries per second registered during a minute.
Minimum	The lowest number of search queries per second registered during a minute.
Sum	The sum of all queries executed within the minute.

For example, within one minute, you might have a pattern like this: one second of high load that is the maximum for SearchQueriesPerSecond, followed by 58 seconds of average load, and finally one second with only one query, which is the minimum.

Another example: if a node emits 100 metrics, where the value of each metric is 40, then "Count" is 100, "Sum" is 4000, "Average" is 40, and "Max" is 40.

Search latency

Search latency indicates how long a query takes to complete.

[Expand table](#)

Aggregation type	Latency
Average	Average query duration in milliseconds.
Count	The number of metrics emitted to the log within the one-minute interval.
Maximum	Longest running query in the sample.
Minimum	Shortest running query in the sample.
Total	Total execution time of all queries in the sample, executing within the interval (one minute).

Throttled search queries percentage

This metric refers to queries that are dropped instead of processed. Throttling occurs when the number of requests in execution exceed capacity. You might see an increase in throttled requests when a replica is taken out of rotation or during indexing. Both query and indexing requests are handled by the same set of resources.

The service determines whether to drop requests based on resource consumption. The percentage of resources consumed across memory, CPU, and disk IO are averaged over a period of time. If this

percentage exceeds a threshold, all requests to the index are throttled until the volume of requests is reduced.

Depending on your client, a throttled request is indicated in these ways:

- A service returns an error "You are sending too many requests. Please try again later."
- A service returns a 503 error code indicating the service is currently unavailable.
- If you're using the Azure portal (for example, Search Explorer), the query is dropped silently and you need to select **Search** again.

To confirm throttled queries, use **Throttled search queries** metric. You can explore metrics in the Azure portal or create an alert metric as described in this article. For queries that were dropped within the sampling interval, use *Total* to get the percentage of queries that didn't execute.

[+] [Expand table](#)

Aggregation type	Throttling
Average	Percentage of queries dropped within the interval.
Count	The number of metrics emitted to the log within the one-minute interval.
Maximum	Percentage of queries dropped within the interval.
Minimum	Percentage of queries dropped within the interval.
Total	Percentage of queries dropped within the interval.

For **Throttled Search Queries Percentage**, minimum, maximum, average and total, all have the same value: the percentage of search queries that were throttled, from the total number of search queries during one minute.

Metric dimensions

For information about what metric dimensions are, see [Multi-dimensional metrics](#).

Azure AI Search has dimensions associated with the following metrics that capture a count of documents or skills that were executed.

[+] [Expand table](#)

Metric name	Description	Dimensions	Sample use cases
Document processed count	Shows the number of indexer processed documents.	Data source name, failed, index name, indexer name, skillset name	Can be referenced as a rough measure of throughput (number of documents processed by indexer over time) - Set up to alert on failed documents
Skill execution invocation count	Shows the number of skill invocations.	Data source name, failed, index name, indexer name,	Reference to ensure skills are invoked as expected by comparing relative invocation numbers between skills and number of skill

Metric name	Description	Dimensions	Sample use cases
		skill name, skill type, skillset name	invocations to the number of documents. - Set up to alert on failed skill invocations

[+] [Expand table](#)

Dimension name	Description
DataSourceName	A named data source connection used during indexer execution. Valid values are one of the supported data source types .
Failed	Indicates whether the instance failed.
IndexerName	Name of an indexer.
IndexName	Name of an index.
SkillsetName	Name of a skillset used during indexer execution.
SkillName	Name of a skill within a skillset.
SkillType	The @odata.type of the skill.

Resource logs

This section lists the types of resource logs you can collect for this service. The section pulls from the list of [all resource logs category types supported in Azure Monitor](#).

Supported resource logs for Microsoft.Search/searchServices

[+] [Expand table](#)

Category	Category display name	Log table	Supports basic log plan	Supports ingestion-time transformation	Example queries	Costs to export
OperationLogs	Operation Logs	AzureDiagnostics Logs from multiple Azure resources.	No	No		No

Azure Monitor Logs tables

This section lists the Azure Monitor Logs tables relevant to this service, which are available for query by Log Analytics using Kusto queries. The tables contain resource log data and possibly more depending on what is collected and routed to them.

Search Services

[] Expand table

Table	Description
AzureActivity	Entries from the Azure activity log provide insight into control plane operations. Tasks invoked on the control plane, such as adding or removing replicas and partitions, are represented through a "Get Admin Key" activity.
AzureDiagnostics	Logged query and indexing operations. Queries against the AzureDiagnostics table in Log Analytics can include the common properties, the search-specific properties , and the search-specific operations listed in the schema reference section.
AzureMetrics	Metric data emitted by Azure AI Search that measures health and performance.

Resource log tables

The following table lists the properties of resource logs in Azure AI Search. The resource logs are collected into Azure Monitor Logs or Azure Storage. In Azure Monitor, logs are collected in the AzureDiagnostics table under the resource provider name of `Microsoft.Search`.

[] Expand table

Azure Storage field or property	Azure Monitor Logs property	Description
time	TIMESTAMP	The date and time (UTC) when the operation occurred.
resourceId	Concat("/", "/subscriptions", SubscriptionId, "resourceGroups", ResourceGroupName, "providers/Microsoft.Search/searchServices", ServiceName)	The Azure AI Search resource for which logs are enabled.
category	"OperationLogs"	Log categories include <code>Audit</code> , <code>Operational</code> , <code>Execution</code> , and <code>Request</code> .
operationName	Name	Name of the operation. The operation name can be <code>Indexes.ListIndexStatsSummaries</code> , <code>Indexes.Get</code> , <code>Indexes.Stats</code> , <code>Indexers.List</code> , <code>Query.Search</code> , <code>Query.Suggest</code> , <code>Query.Lookup</code> , <code>Query.Autocomplete</code> , <code>CORS.Preflight</code> , <code>Indexes.Update</code> , <code>Indexes.Prototype</code> , <code>ServiceStats</code> , <code>DataSources.List</code> , <code>Indexers.Warmup</code> .
durationMS	DurationMilliseconds	The duration of the operation, in milliseconds.
operationVersion	ApiVersion	The API version used on the request.
resultType	(Failed) ? "Failed" : "Success"	The type of response.

Azure Storage field or property	Azure Monitor Logs property	Description
resultSignature	Status	The HTTP response status of the operation.
properties	Properties	Any extended properties related to this category of events.

Activity log

The linked table lists the operations that can be recorded in the activity log for this service. These operations are a subset of [all the possible resource provider operations in the activity log](#).

For more information on the schema of activity log entries, see [Activity Log schema](#).

The following table lists common operations related to Azure AI Search that may be recorded in the activity log. For a complete listing of all Microsoft.Search operations, see [Microsoft.Search resource provider operations](#).

[] [Expand table](#)

Operation	Description
Get Admin Key	Any operation that requires administrative rights is logged as a "Get Admin Key" operation.
Get Query Key	Any read-only operation against the documents collection of an index.
Regenerate Admin Key	A request to regenerate either the primary or secondary admin API key.

Common entries include references to API keys - generic informational notifications like *Get Admin Key* and *Get Query keys*. These activities indicate requests that were made using the admin key (create or delete objects) or query key, but don't show the request itself. For information of this grain, you must configure resource logging.

Alternatively, you might gain some insight through change history. In the Azure portal, select the activity to open the detail page and then select "Change history" for information about the underlying operation.

Other schemas

The following schemas are in use for this service.

If you're building queries or custom reports, the data structures that contain Azure AI Search resource logs conform to the following schemas.

For resource logs sent to blob storage, each blob has one root object called **records** containing an array of log objects. Each blob contains records for all the operations that took place during the same hour.

Resource log schema

All resource logs available through Azure Monitor share a [common top-level schema](#). Azure AI Search supplements with [more properties](#) and [operations](#) that are unique to a search service.

The following example illustrates a resource log that includes common properties (TimeGenerated, Resource, Category, and so forth) and search-specific properties (OperationName and OperationVersion).

[+] [Expand table](#)

Name	Type	Description and example
TimeGenerated	Datetime	Timestamp of the operation. For example: <code>2021-12-07T00:00:43.6872559Z</code>
Resource	String	Resource ID. For example: <code>/subscriptions/<your-subscription-id>/resourceGroups/<your-resource-group-name>/providers/Microsoft.Search/searchServices/<your-search-service-name></code>
Category	String	"OperationLogs". This value is a constant. OperationLogs is the only category used for resource logs.
OperationName	String	The name of the operation (see the full list of operations). An example is <code>Query.Search</code>
OperationVersion	String	The api-version used on the request. For example: <code>2024-07-01</code>
ResultType	String	"Success". Other possible values: Success or Failure
ResultSignature	Int	An HTTP result code. For example: <code>200</code>
DurationMS	Int	Duration of the operation in milliseconds.
Properties	Object	Object containing operation-specific data. See the following properties schema table.

Properties schema

The following properties are specific to Azure AI Search.

[+] [Expand table](#)

Name	Type	Description and example
Description_s	String	The operation's endpoint. For example: <code>GET /indexes('content')/docs</code>
Documents_d	Int	Number of documents processed.
IndexName_s	String	Name of the index associated with the operation.
Query_s	String	The query parameters used in the request. For example: <code>?search=beach access\$count=true&api-version=2024-07-01</code>

OperationName values (logged operations)

The following operations can appear in a resource log.

OperationName	Description
DataSources.*	Applies to indexer data sources. Can be Create, Delete, Get, List.
DebugSessions.*	Applies to a debug session. Can be Create, Delete, Get, List, Start, and Status.
DebugSessions.DocumentStructure	An enriched document is loaded into a debug session.
DebugSessions.RetrieveIndexerExecutionHistoricalData	A request for indexer execution details.
DebugSessions.RetrieveProjectedIndexerExecutionHistoricalData	Execution history for enrichments projected to a knowledge store.
Indexers.*	Applies to an indexer. Can be Create, Delete, Get, List, and Status.
Indexes.*	Applies to a search index. Can be Create, Delete, Get, List.
indexes.Prototype	This index is created by the Import Data wizard.
Indexing.Index	This operation is a call to Index Documents .
Metadata.GetMetadata	A request for search service system data.
Query.Autocomplete	An autocomplete query against an index. See Query types and composition .
Query.Lookup	A lookup query against an index. See Query types and composition .
Query.Search	A full text search request against an index. See Query types and composition .
Query.Suggest	Type ahead query against an index. See Query types and composition .
ServiceStats	This operation is a routine call to Get Service Statistics , either called directly or implicitly to populate a portal overview page when it's loaded or refreshed.
Skillsets.*	Applies to a skillset. Can be Create, Delete, Get, List.

Related content

- See [Monitor Azure AI Search](#) for a description of monitoring Azure AI Search.
- See [Monitor Azure resources with Azure Monitor](#) for details on monitoring Azure resources.

Azure Policy Regulatory Compliance controls for Azure AI Search

08/18/2025

If you are using [Azure Policy](#) to enforce the recommendations in [Microsoft cloud security benchmark](#), then you probably already know that you can create policies for identifying and fixing non-compliant services. These policies might be custom, or they might be based on built-in definitions that provide compliance criteria and appropriate solutions for well-understood best practices.

For Azure AI Search, there is currently one built-definition, listed below, that you can use in a policy assignment. The built-in is for logging and monitoring. By using this built-in definition in a [policy that you create](#), the system will scan for search services that do not have [resource logging](#), and then enable it accordingly.

[Regulatory Compliance in Azure Policy](#) provides Microsoft-created and managed initiative definitions, known as *built-ins*, for the **compliance domains** and **security controls** related to different compliance standards. This page lists the **compliance domains** and **security controls** for Azure AI Search. You can assign the built-ins for a **security control** individually to help make your Azure resources compliant with the specific standard.

The title of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Policy Version** column to view the source on the [Azure Policy GitHub repo](#).

Important

Each control is associated with one or more [Azure Policy](#) definitions. These policies might help you [assess compliance](#) with the control. However, there often isn't a one-to-one or complete match between a control and one or more policies. As such, **Compliant** in Azure Policy refers only to the policies themselves. This doesn't ensure that you're fully compliant with all requirements of a control. In addition, the compliance standard includes controls that aren't addressed by any Azure Policy definitions at this time. Therefore, compliance in Azure Policy is only a partial view of your overall compliance status. The associations between controls and Azure Policy Regulatory Compliance definitions for these compliance standards can change over time.

CIS Microsoft Azure Foundations Benchmark 1.3.0

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - CIS Microsoft Azure Foundations Benchmark 1.3.0](#). For more information about this compliance standard, see [CIS Microsoft Azure Foundations Benchmark](#).

[] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
5 Logging and Monitoring	5.3	Ensure that Diagnostic Logs are enabled for all services which support it.	Resource logs in Search services should be enabled	5.0.0 ↗

CIS Microsoft Azure Foundations Benchmark 1.4.0

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for CIS v1.4.0](#). For more information about this compliance standard, see [CIS Microsoft Azure Foundations Benchmark](#).

[] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
5 Logging and Monitoring	5.3	Ensure that Diagnostic Logs Are Enabled for All Services that Support it.	Resource logs in Search services should be enabled	5.0.0 ↗

CIS Microsoft Azure Foundations Benchmark 2.0.0

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for CIS v2.0.0](#). For more information about this compliance standard, see [CIS Microsoft Azure Foundations Benchmark](#).

[] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
5	5.4	Ensure that Azure Monitor Resource Logging is Enabled for All Services that Support it	Resource logs in Search services should be enabled	5.0.0 ↗

CMMC Level 3

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - CMMC Level 3](#). For more information about this compliance standard, see [Cybersecurity Maturity Model Certification \(CMMC\)](#).

[Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC.1.001	Limit information system access to authorized users, processes acting on behalf of authorized users, and devices (including other information systems).	Azure AI Services resources should restrict network access	3.3.0 ↗
Access Control	AC.1.002	Limit information system access to the types of transactions and functions that authorized users are permitted to execute.	Azure AI Services resources should restrict network access	3.3.0 ↗
Access Control	AC.2.016	Control the flow of CUI in accordance with approved authorizations.	Azure AI Services resources should restrict network access	3.3.0 ↗
Configuration Management	CM.3.068	Restrict, disable, or prevent the use of nonessential programs, functions, ports, protocols, and services.	Azure AI Services resources should restrict network access	3.3.0 ↗
System and Communications Protection	SC.1.175	Monitor, control, and protect communications (i.e., information transmitted or received by organizational systems) at the external	Azure AI Services resources	3.3.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
		boundaries and key internal boundaries of organizational systems.	should restrict network access ↗	
System and Communications Protection	SC.3.183	Deny network communications traffic by default and allow network communications traffic by exception (i.e., deny all, permit by exception).	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

FedRAMP High

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - FedRAMP High](#). For more information about this compliance standard, see [FedRAMP High](#) ↗.

[] Expand table

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2	Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (1)	Automated System Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (7)	Role-Based Schemes	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-3	Access Enforcement	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search services should disable public network access ↗	1.0.1 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
Access Control	AC-17	Remote Access	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Audit And Accountability	AU-6 (4)	Central Review And Analysis	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-6 (5)	Integration / Scanning And Monitoring Capabilities	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-12	Audit Generation	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-12 (1)	System-Wide / Time-Correlated Audit Trail	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Identification And Authentication	IA-2	Identification And Authentication (Organizational Users)	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification And Authentication	IA-4	Identifier Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

FedRAMP Moderate

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - FedRAMP Moderate](#). For more information about this compliance standard, see [FedRAMP Moderate \[↗\]\(#\)](#).

[\[\]](#) Expand table

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2	Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (1)	Automated System Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (7)	Role-Based Schemes	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-3	Access Enforcement	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search services should disable public network access ↗	1.0.1 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
Access Control	AC-17	Remote Access	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Audit And Accountability	AU-12	Audit Generation	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Identification And Authentication	IA-2	Identification And Authentication (Organizational Users)	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification And Authentication	IA-4	Identifier Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

HIPAA HITRUST

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - HIPAA HITRUST](#). For more information about this compliance standard, see [HIPAA HITRUST](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
12 Audit Logging & Monitoring	1208.09aa3System.1-09.aa	1208.09aa3System.1-09.aa 09.10 Monitoring	Resource logs in Search services should be enabled	5.0.0

Microsoft cloud security benchmark

The [Microsoft cloud security benchmark](#) provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Microsoft cloud security benchmark, see the [Azure Security Benchmark mapping files](#).

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - Microsoft cloud security benchmark](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Network Security	NS-2	Secure cloud services with network controls	Azure AI Services resources should restrict network access	3.3.0
Network Security	NS-2	Secure cloud services with network controls	Azure AI Services resources should use Azure Private Link	1.0.0
Identity Management	IM-1	Use centralized identity and authentication system	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
Logging and Threat Detection	LT-3	Enable logging for security investigation	Diagnostic logs in Azure AI services resources should be enabled	1.0.0
Logging and Threat Detection	LT-3	Enable logging for security investigation	Resource logs in Search services should be enabled	5.0.0

NIST SP 800-171 R2

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - NIST SP 800-171 R2](#). For more information about this compliance standard, see [NIST SP 800-171 R2](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	3.1.1	Limit system access to authorized users, processes acting on behalf of authorized users, and devices (including other systems).	Azure AI Search service should use a SKU that supports private link	1.0.1
Access Control	3.1.1	Limit system access to authorized users, processes acting on behalf of authorized users, and devices (including other systems).	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
Access Control	3.1.12	Monitor and control remote access sessions.	Azure AI Search service should use a SKU that supports private link	1.0.1
Access Control	3.1.13	Employ cryptographic mechanisms to protect the confidentiality of remote access sessions.	Azure AI Search service should use a SKU that supports private link	1.0.1
Access Control	3.1.14	Route remote access via managed access control points.	Azure AI Search service should use a SKU that supports private link	1.0.1
Access Control	3.1.2	Limit system access to the types of transactions and functions that authorized users are permitted to execute.	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
Access Control	3.1.3	Control the flow of CUI in accordance with approved authorizations.	Azure AI Search service should use a SKU that supports private link	1.0.1
Access Control	3.1.3	Control the flow of CUI in accordance with approved authorizations.	Azure AI Search services should disable public network access	1.0.1

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	3.1.3	Control the flow of CUI in accordance with approved authorizations.	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System and Communications Protection	3.13.1	Monitor, control, and protect communications (i.e., information transmitted or received by organizational systems) at the external boundaries and key internal boundaries of organizational systems.	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System and Communications Protection	3.13.1	Monitor, control, and protect communications (i.e., information transmitted or received by organizational systems) at the external boundaries and key internal boundaries of organizational systems.	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System and Communications Protection	3.13.1	Monitor, control, and protect communications (i.e., information transmitted or received by organizational systems) at the external boundaries and key internal boundaries of organizational systems.	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System and Communications Protection	3.13.2	Employ architectural designs, software development techniques, and systems engineering principles that promote effective information security within organizational systems.	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System and Communications Protection	3.13.2	Employ architectural designs, software development techniques, and systems engineering principles that promote effective information security within organizational systems.	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System and Communications Protection	3.13.2	Employ architectural designs, software development techniques, and systems engineering principles that promote effective	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
		information security within organizational systems.		
System and Communications Protection	3.13.5	Implement subnetworks for publicly accessible system components that are physically or logically separated from internal networks.	Azure AI Search service should use a SKU that supports private link	1.0.1 ↗
System and Communications Protection	3.13.5	Implement subnetworks for publicly accessible system components that are physically or logically separated from internal networks.	Azure AI Search services should disable public network access	1.0.1 ↗
System and Communications Protection	3.13.5	Implement subnetworks for publicly accessible system components that are physically or logically separated from internal networks.	Azure AI Services resources should restrict network access	3.3.0 ↗
System and Communications Protection	3.13.6	Deny network communications traffic by default and allow network communications traffic by exception (i.e., deny all, permit by exception).	Azure AI Search services should disable public network access	1.0.1 ↗
System and Communications Protection	3.13.6	Deny network communications traffic by default and allow network communications traffic by exception (i.e., deny all, permit by exception).	Azure AI Services resources should restrict network access	3.3.0 ↗
Audit and Accountability	3.3.1	Create and retain system audit logs and records to the extent needed to enable the monitoring, analysis, investigation, and reporting of unlawful or unauthorized system activity	Resource logs in Search services should be enabled	5.0.0 ↗
Audit and Accountability	3.3.2	Ensure that the actions of individual system users can be uniquely traced to those users, so they can be held accountable for their actions.	Resource logs in Search services should be enabled	5.0.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Identification and Authentication	3.5.1	Identify system users, processes acting on behalf of users, and devices.	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification and Authentication	3.5.2	Authenticate (or verify) the identities of users, processes, or devices, as a prerequisite to allowing access to organizational systems.	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification and Authentication	3.5.5	Prevent reuse of identifiers for a defined period.	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification and Authentication	3.5.6	Disable identifiers after a defined period of inactivity.	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗

NIST SP 800-53 Rev. 4

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 4](#). For more information about this compliance standard, see [NIST SP 800-53 Rev. 4](#) ↗.

Expand table

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2	Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (1)	Automated System Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2 (7)	Role-Based Schemes	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-3	Access Enforcement	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search services should disable public network access ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
Access Control	AC-17 (1)	Remote Access	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Audit And Accountability	AU-6 (4)	Central Review And Analysis	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-6 (5)	Integration / Scanning And Monitoring Capabilities	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-12	Audit Generation	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit And Accountability	AU-12 (1)	System-Wide / Time-Correlated Audit Trail	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Identification And Authentication	IA-2	Identification And Authentication (Organizational Users)	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification And Authentication	IA-4	Identifier Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7	Boundary Protection	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System And Communications Protection	SC-7 (3)	Access Points	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

NIST SP 800-53 Rev. 5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 5](#). For more information about this compliance standard, see [NIST SP 800-53 Rev. 5](#).

[] Expand table

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2	Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-2 (1)	Automated System Account Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-2 (7)	Privileged User Accounts	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-3	Access Enforcement	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Search services should disable public network access ↗	1.0.1 ↗
Access Control	AC-4	Information Flow Enforcement	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
Access Control	AC-17 (1)	Remote Access	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Access Control	AC-17 (1)	Monitoring and Control	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
Audit and Accountability	AU-6 (4)	Central Review and Analysis	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit and Accountability	AU-6 (5)	Integrated Analysis of Audit Records	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit and Accountability	AU-12	Audit Record Generation	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Audit and Accountability	AU-12 (1)	System-wide and Time-correlated Audit Trail	Resource logs in Search services should be enabled ↗	5.0.0 ↗
Identification and Authentication	IA-2	Identification and Authentication (organizational Users)	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗
Identification and Authentication	IA-4	Identifier Management	Azure AI Services resources should have key access disabled (disable local authentication) ↗	1.1.0 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
System and Communications Protection	SC-7	Boundary Protection	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System and Communications Protection	SC-7	Boundary Protection	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System and Communications Protection	SC-7	Boundary Protection	Azure AI Services resources should restrict network access ↗	3.3.0 ↗
System and Communications Protection	SC-7 (3)	Access Points	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
System and Communications Protection	SC-7 (3)	Access Points	Azure AI Search services should disable public network access ↗	1.0.1 ↗
System and Communications Protection	SC-7 (3)	Access Points	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

NL BIO Cloud Theme

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for NL BIO Cloud Theme](#). For more information about this compliance standard, see [Baseline Information Security Government Cybersecurity - Digital Government \(digitaleoverheid.nl\)](#) ↗ .

[] Expand table

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
U.07.1 Data separation - Isolated	U.07.1	Permanent isolation of data is a multi-tenant architecture. Patches are realized in a controlled manner.	Azure AI Search service should use a SKU that supports private link ↗	1.0.1 ↗
U.07.1 Data separation -	U.07.1	Permanent isolation of data is a multi-tenant architecture.	Azure AI Search services should disable public	1.0.1 ↗

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Isolated		Patches are realized in a controlled manner.	network access	
U.07.1 Data separation - Isolated	U.07.1	Permanent isolation of data is a multi-tenant architecture. Patches are realized in a controlled manner.	Azure AI Services resources should restrict network access	3.3.0
U.07.3 Data separation - Management features	U.07.3	U.07.3 - The privileges to view or modify CSC data and/or encryption keys are granted in a controlled manner and use is logged.	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
U.10.2 Access to IT services and data - Users	U.10.2	Under the responsibility of the CSP, access is granted to administrators.	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
U.10.3 Access to IT services and data - Users	U.10.3	Only users with authenticated equipment can access IT services and data.	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
U.10.5 Access to IT services and data - Competent	U.10.5	Access to IT services and data is limited by technical measures and has been implemented.	Azure AI Services resources should have key access disabled (disable local authentication)	1.1.0
U.15.1 Logging and monitoring - Events logged	U.15.1	The violation of the policy rules is recorded by the CSP and the CSC.	Resource logs in Search services should be enabled	5.0.0

Reserve Bank of India IT Framework for Banks v2016

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - RBI ITF Banks v2016](#). For more information about this compliance standard, see [RBI ITF Banks v2016 \(PDF\)](#).

[Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Anti-Phishing		Anti-Phishing-14.1	Azure AI Services resources should restrict network access ↗	3.3.0 ↗

RMIT Malaysia

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - RMIT Malaysia](#). For more information about this compliance standard, see [RMIT Malaysia ↗](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Security of Digital Services	10.66	Security of Digital Services - 10.66	Deploy Diagnostic Settings for Search Services to Event Hub ↗	2.0.0 ↗
Security of Digital Services	10.66	Security of Digital Services - 10.66	Deploy Diagnostic Settings for Search Services to Log Analytics workspace ↗	1.0.0 ↗

Spain ENS

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for Spain ENS](#). For more information about this compliance standard, see [CCN-STIC 884 ↗](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Operational framework	op.exp.7	Operation	Resource logs in Search services should be enabled ↗	5.0.0 ↗

SWIFT CSP-CSCF v2021

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for SWIFT CSP-CSCF v2021](#). For more information about this compliance standard, see [SWIFT CSP CSCF v2021](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Detect Anomalous Activity to Systems or Transaction Records	6.4	Logging and Monitoring	Resource logs in Search services should be enabled	5.0.0

SWIFT CSP-CSCF v2022

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance details for SWIFT CSP-CSCF v2022](#). For more information about this compliance standard, see [SWIFT CSP CSCF v2022](#).

[+] [Expand table](#)

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
6. Detect Anomalous Activity to Systems or Transaction Records	6.4	Record security events and detect anomalous actions and operations within the local SWIFT environment.	Resource logs in Search services should be enabled	5.0.0

Next steps

- Learn more about [Azure Policy Regulatory Compliance](#).
- See the built-ins on the [Azure Policy GitHub repo](#).

Azure security baseline for Azure AI Search

Article • 02/25/2025

This security baseline applies guidance from the [Microsoft cloud security benchmark version 1.0](#) to Azure AI Search. The Microsoft cloud security benchmark provides recommendations on how you can secure your cloud solutions on Azure. The content is grouped by the security controls defined by the Microsoft cloud security benchmark and the related guidance applicable to Azure Cognitive Search.

You can monitor this security baseline and its recommendations using Microsoft Defender for Cloud. Azure Policy definitions will be listed in the Regulatory Compliance section of the Microsoft Defender for Cloud portal page.

When a feature has relevant Azure Policy Definitions, they are listed in this baseline to help you measure compliance with the Microsoft cloud security benchmark controls and recommendations. Some recommendations may require a paid Microsoft Defender plan to enable certain security scenarios.

ⓘ Note

Features not applicable to Azure AI Search have been excluded. To see how Azure AI Search completely maps to the Microsoft cloud security benchmark, see the [full Azure AI Search security baseline mapping file ↗](#).

Security profile

The security profile summarizes high-impact behaviors of Azure AI Search, which may result in increased security considerations.

ⓘ [Expand table](#)

Service Behavior Attribute	Value
Product Category	AI+ML, Mobile, Web
Customer can access HOST / OS	No Access
Service can be deployed into customer's virtual network	False
Stores customer content at rest	True

Network security

For more information, see the [Microsoft cloud security benchmark: Network security](#).

NS-1: Establish network segmentation boundaries

Features

Virtual Network Integration

Description: Service supports deployment into customer's private Virtual Network (VNet).

[Learn more.](#)

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Network Security Group Support

Description: Service network traffic respects Network Security Groups rule assignment on its subnets. [Learn more.](#)

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

NS-2: Secure cloud services with network controls

Features

Azure Private Link

Description: Service native IP filtering capability for filtering network traffic (not to be confused with NSG or Azure Firewall). [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: For outbound connections through a private endpoint, refer to: [Make outbound connections through a private endpoint](#)

Configuration Guidance: Deploy private endpoints to establish a private access point for the resources. Block all connections on the public endpoint for your search service. Increase security for the virtual network, by enabling you to block exfiltration of data from the virtual network.

Reference: [Create a Private Endpoint for a secure connection to Azure AI Search](#)

Disable Public Network Access

Description: Service supports disabling public network access either through using service-level IP ACL filtering rule (not NSG or Azure Firewall) or using a 'Disable Public Network Access' toggle switch. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Azure AI Search supports IP rules for inbound access through a firewall, similar to the IP rules you'll find in an Azure virtual network security group. By leveraging IP rules, you can restrict search service access to an approved set of machines and cloud services. Access to data stored in your search service from the approved sets of machines and services will still require the caller to present a valid authorization token.

Reference: [Configure an IP firewall for Azure AI Search](#)

Identity management

For more information, see the [Microsoft cloud security benchmark: Identity management](#).

IM-1: Use centralized identity and authentication system

Features

Azure AD Authentication Required for Data Plane Access

Description: Service supports using Azure AD authentication for data plane access. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Local Authentication Methods for Data Plane Access

Description: Local authentications methods supported for data plane access, such as a local username and password. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Avoid the usage of local authentication methods or accounts, these should be disabled wherever possible. Instead use Azure AD to authenticate where possible.

Reference: [Use roles for Azure AI Search authentication](#)

IM-3: Manage application identities securely and automatically

Features

Managed Identities

Description: Data plane actions support authentication using managed identities. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure managed identities instead of service principals when possible, which can authenticate to Azure services and resources that support Azure Active Directory (Azure AD) authentication. Managed identity credentials are fully managed, rotated, and protected by the platform, avoiding hard-coded credentials in source code or configuration files.

Reference: [Authorize access to a search app using Azure Active Directory](#)

Service Principals

Description: Data plane supports authentication using service principals. [Learn more](#).

 [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: There is no current Microsoft guidance for this feature configuration. Please review and determine if your organization wants to configure this security feature.

IM-7: Restrict resource access based on conditions

Features

Conditional Access for Data Plane

Description: Data plane access can be controlled using Azure AD Conditional Access Policies. [Learn more](#).

 [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Define the applicable conditions and criteria for Azure Active Directory (Azure AD) conditional access in the workload. Consider common use cases such as

blocking or granting access from specific locations, blocking risky sign-in behavior, or requiring organization-managed devices for specific applications.

IM-8: Restrict the exposure of credential and secrets

Features

Service Credential and Secrets Support Integration and Storage in Azure Key Vault

Description: Data plane supports native use of Azure Key Vault for credential and secrets store. [Learn more.](#)

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Privileged access

For more information, see the [Microsoft cloud security benchmark: Privileged access](#).

PA-1: Separate and limit highly privileged/administrative users

Features

Local Admin Accounts

Description: Service has the concept of a local administrative account. [Learn more.](#)

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

PA-7: Follow just enough administration (least privilege) principle

Features

Azure RBAC for Data Plane

Description: Azure Role-Based Access Control (Azure RBAC) can be used to manage access to service's data plane actions. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Azure provides a global role-based access control (RBAC) authorization system for all services running on the platform. In AI Search, you can use Azure roles for:

- Control plane operations (service administration tasks through Azure Resource Manager).
- Data plane operations, such as creating, loading, and querying indexes.

Reference: [Use Azure role-based access controls \(Azure RBAC\) in Azure AI Search](#)

PA-8: Determine access process for cloud provider support

Features

Customer Lockbox

Description: Customer Lockbox can be used for Microsoft support access. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: In support scenarios where Microsoft needs to access your data, use Customer Lockbox to review, then approve or reject each of Microsoft's data access requests.

Data protection

For more information, see the [Microsoft cloud security benchmark: Data protection](#).

DP-1: Discover, classify, and label sensitive data

Features

Sensitive Data Discovery and Classification

Description: Tools (such as Azure Purview or Azure Information Protection) can be used for data discovery and classification in the service. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

DP-2: Monitor anomalies and threats targeting sensitive data

Features

Data Leakage/Loss Prevention

Description: Service supports DLP solution to monitor sensitive data movement (in customer's content). [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

DP-3: Encrypt sensitive data in transit

Features

Data in Transit Encryption

Description: Service supports data in-transit encryption for data plane. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: [Azure AI Search data in transit encryption](#)

DP-4: Enable data at rest encryption by default

Features

Data at Rest Encryption Using Platform Keys

Description: Data at-rest encryption using platform keys is supported, any customer content at rest is encrypted with these Microsoft managed keys. [Learn more](#).

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: [Azure AI Search default data encryption using service-managed keys](#)

DP-5: Use customer-managed key option in data at rest encryption when required

Features

Data at Rest Encryption Using CMK

Description: Data at-rest encryption using customer-managed keys is supported for customer content stored by the service. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: If required for regulatory compliance, define the use case and service scope where encryption using customer-managed keys are needed. Enable and implement data at rest encryption using customer-managed key for those services.

Reference: [Configure customer-managed keys for data encryption in Azure AI Search](#)

DP-6: Use a secure key management process

Features

Key Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer keys, secrets, or certificates. [Learn more.](#)

[+] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure Key Vault to create and control the life cycle of your encryption keys, including key generation, distribution, and storage. Rotate and revoke your keys in Azure Key Vault and your service based on a defined schedule or when there is a key retirement or compromise. When there is a need to use customer-managed key (CMK) in the workload, service, or application level, ensure you follow the best practices for key management: Use a key hierarchy to generate a separate data encryption key (DEK) with your key encryption key (KEK) in your key vault. Ensure keys are registered with Azure Key Vault and referenced via key IDs from the service or application. If you need to bring your own key (BYOK) to the service (such as importing HSM-protected keys from your on-premises HSMs

into Azure Key Vault), follow recommended guidelines to perform initial key generation and key transfer.

Reference: [Configure customer-managed keys for data encryption in Azure AI Search](#)

DP-7: Use a secure certificate management process

Features

Certificate Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer certificates.
[Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Asset management

For more information, see the [Microsoft cloud security benchmark: Asset management](#).

AM-2: Use only approved services

Features

Azure Policy Support

Description: Service configurations can be monitored and enforced via Azure Policy. [Learn more.](#)

[+] Expand table

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Microsoft Defender for Cloud to configure Azure Policy to audit and enforce configurations of your Azure resources. Use Azure Monitor to create alerts when there is a configuration deviation detected on the resources. Use Azure Policy [deny] and [deploy if not exists] effects to enforce secure configuration across Azure resources.

Reference: [Azure Ai Search Policies](#)

Logging and threat detection

For more information, see the [Microsoft cloud security benchmark: Logging and threat detection](#).

LT-1: Enable threat detection capabilities

Features

Microsoft Defender for Service / Product Offering

Description: Service has an offering-specific Microsoft Defender solution to monitor and alert on security issues. [Learn more](#).

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

LT-4: Enable logging for security investigation

Features

Azure Resource Logs

Description: Service produces resource logs that can provide enhanced service-specific metrics and logging. The customer can configure these resource logs and send them to their own data sink like a storage account or log analytics workspace. [Learn more](#).

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Enable resource logs for the service to view the Azure AI Search operations logs, search metrics, and etc.

Reference: [Azure AI Search resource log](#)

Backup and recovery

For more information, see the [Microsoft cloud security benchmark: Backup and recovery](#).

BR-1: Ensure regular automated backups

Features

Azure Backup

Description: The service can be backed up by the Azure Backup service. [Learn more](#).

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Service Native Backup Capability

Description: Service supports its own native backup capability (if not using Azure Backup).

[Learn more](#).

[] [Expand table](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Feature notes: Because Azure AI Search isn't a primary data storage solution, Microsoft doesn't provide a formal mechanism for self-service backup and restore. However, you can backup and

restore the index using your own code. Refer to: [Back up and restore alternatives](#)

Configuration Guidance: This feature is not supported to secure this service.

Next steps

- See the [Microsoft cloud security benchmark overview](#)
- Learn more about [Azure security baselines](#)

Skills for extra processing during indexing (Azure AI Search)

This article describes the skills in Azure AI Search that you can include in a [skillset](#) to access external processing.

A *skill* is an atomic operation that transforms content in some way. Often, it's an operation that recognizes or extracts text, but it can also be a utility skill that reshapes existing enrichments. The output is usually text-based for use in [full-text search](#) or vectors for use in [vector search](#).

Skills are organized into the following categories:

- A *built-in skill* wraps API calls to another Azure resource, where the inputs, outputs, and processing steps are well understood. Some built-in skills require an attached resource solely for billing, while others use your Azure-hosted model or resource for both billing and processing.
- A *custom skill* provides custom code that executes externally to the search service. It's accessed through a URI. Custom code is often made available through an Azure function app. To attach an open-source or third-party vectorization model, use a custom skill.
- A *utility skill* is internal to Azure AI Search, with no dependency on external resources or outbound connections. Most utility skills are nonbillable.

Built-in skills

There are two types of built-in skills:

- Skills that connect to a [Microsoft Foundry resource](#) (for billing only)
- Skills that connect to an [Azure-hosted model or resource](#) (for billing and processing)

Foundry resource

Skills in this category call subservices of Foundry Tools. For billing rather than processing, you must [attach a Foundry resource to your skillset](#). Azure AI Search uses internal resources to execute these skills and only uses your Foundry resource for billing purposes.

A small quantity of processing is nonbillable, but at larger volumes, processing is billable. These skills are based on pretrained models from Foundry Tools, which means you can't train the models using your own data.

These skills are billed at the Standard rate.

Skill	Description	Metered by
Azure Vision multimodal embeddings	Multimodal image and text vectorization.	Foundry Tools (pricing ↗)
Custom Entity Lookup	Looks for text from a custom, user-defined list of words and phrases.	Azure AI Search (pricing ↗)
Entity Linking	This skill uses a pretrained model to generate links for recognized entities to articles in Wikipedia.	Foundry Tools (pricing ↗)
Entity Recognition	This skill uses a pretrained model to establish entities for a fixed set of categories: "Person", "Location", "Organization", "Quantity", "DateTime", "URL", "Email", "PersonType", "Event", "Product", "Skill", "Address", "Phone Number" and "IP Address" fields.	Foundry Tools (pricing ↗)
Image Analysis	This skill uses an image detection algorithm to identify the content of an image and generate a text description.	Foundry Tools (pricing ↗)
Key Phrase Extraction	This skill uses a pretrained model to detect important phrases based on term placement, linguistic rules, proximity to other terms, and how unusual the term is within the source data.	Foundry Tools (pricing ↗)
Language Detection	This skill uses a pretrained model to detect which language is used (one language ID per document). When multiple languages are used within the same text segments, the output is the LCID of the predominantly used language.	Foundry Tools (pricing ↗)
OCR	Optical character recognition.	Foundry Tools (pricing ↗)
PII Detection	This skill uses a pretrained model to extract personal information from a given text. The skill also gives various options for masking the detected personal information entities in the text.	Foundry Tools (pricing ↗)
Sentiment	This skill uses a pretrained model to assign sentiment labels (such as "negative", "neutral" and "positive") based on the highest confidence score found by the service at a sentence and document-level on a record by record basis.	Foundry Tools (pricing ↗)
Text Translation	This skill uses a pretrained model to translate the input text into various languages for normalization or localization use cases.	Foundry Tools (pricing ↗)

Azure-hosted model or resource

Skills in this category call Azure-hosted models or resources that you own for both billing and processing. Although Azure Content Understanding is part of Foundry Tools, the Azure Content Understanding skill connects to your deployed resource for processing, not just billing.

These skills are billed at the Standard rate.

[+] [Expand table](#)

Skill	Description	Metered by
Azure Content Understanding	Connects to Azure Content Understanding for advanced document analysis and semantic chunking.	Azure Content Understanding (pricing ↗)
Azure OpenAI Embedding	Connects to a deployed Azure OpenAI embedding model for integrated vectorization.	Azure OpenAI (pricing ↗)
GenAI Prompt	Extends an AI enrichment pipeline with a Foundry chat completion model.	Azure OpenAI (pricing ↗)

Custom skills

Skills in this category wrap external code that you design, develop, and deploy to the web. You can then call the module from within a skillset as a custom skill.

For guidance on creating a custom skill, see [Define a custom interface](#) and [Example: Creating a custom skill for AI enrichment](#).

[+] [Expand table](#)

Skill	Description	Metered by
AML	Extends an AI enrichment pipeline with a Foundry or Azure Machine Learning model.	None, unless your solution uses a metered Azure service.
Custom Entity Lookup	Extends an AI enrichment pipeline by detecting user-defined entities.	None, unless your solution uses a metered Azure service.
Web API	Extends an AI enrichment pipeline by making an HTTP call into a custom Web API.	None, unless your solution uses a metered Azure service.

Utility skills

Skills in this category execute only on Azure AI Search, iterate mostly on nodes in the enrichment cache, and are mostly nonbillable.

Skill	Description	Metered by
Conditional	Allows filtering, assigning a default value, and merging data based on a condition.	Not applicable
Document Extraction	Extracts content from a file within the enrichment pipeline.	Azure AI Search (pricing ↗) for image extraction
Shaper	Maps output to a complex type (a multi-part data type, which might be used for a full name, a multi-line address, or a combination of last name and a personal identifier.)	Not applicable
Text Merge	Consolidates text from a collection of fields into a single field.	Not applicable
Text Split	Splits text into pages so that you can enrich or augment content incrementally.	Not applicable

Related content

- [Create a skillset](#)
- [Add a custom skill to an AI enrichment pipeline](#)
- [Tutorial: Enriched indexing with AI](#)

Last updated on 11/18/2025

Skill context and input annotation language

07/14/2025

This article is the reference documentation for skill context and input syntax. It's a full description of the expression language used to construct paths to nodes in an enriched document.

Azure AI Search skills can use and [enrich the data coming from the data source and from the output of other skills](#). The data working set that represents the current state of the indexer work for the current document starts from the raw data coming from the data source and is progressively enriched with each skill iteration's output data. That data is internally organized in a tree-like structure that can be queried to be used as skill inputs or to be added to the index. The nodes in the tree can be simple values such as strings and numbers, arrays, or complex objects and even binary files. Even simple values can be enriched with additional structured information. For example, a string can be annotated with additional information that is stored beneath it in the enrichment tree. The expressions used to query that internal structure use a rich syntax that is detailed in this article. The enriched data structure can be [inspected from debug sessions](#). Expressions querying the structure can also be tested from debug sessions.

Throughout the article, we'll use the following enriched data as an example. This data is typical of the kind of structure you would get when enriching a document using a skillset with [OCR](#), [key phrase extraction](#), [text translation](#), [language detection](#), and [entity recognition](#) skills, as well as a custom tokenizer skill.

[\[+\] Expand table](#)

Path	Value
document	
merged_content	"Study of BMN 110 in Pediatric Patients"...
keyphrases	
[0]	"Study of BMN"
[1]	"Syndrome"
[2]	"Pediatric Patients"
...	

Path	Value
<code>locations</code>	
<code>[0]</code>	"IVA"
<code>translated_text</code>	"Étude de BMN 110 chez les patients pédiatriques"...
<code>entities</code>	
<code>[0]</code>	
<code>category</code>	"Organization"
<code>subcategory</code>	<code>null</code>
<code>confidenceScore</code>	0.72
<code>length</code>	3
<code>offset</code>	9
<code>text</code>	"BMN"
...	
<code>organizations</code>	
<code>[0]</code>	"BMN"
<code>language</code>	"en"
<code>normalized_images</code>	
<code>[0]</code>	
<code>layoutText</code>	...
<code>text</code>	
<code>words</code>	
<code>[0]</code>	"Study"
<code>[1]</code>	"of"
<code>[2]</code>	"BMN"
<code>[3]</code>	"110"
...	
<code>[1]</code>	

Path	Value
layoutText	...
text	
words	
[0]	"it"
[1]	"is"
[2]	"certainly"
...	
...	
...	

Document root

All the data is under one root element, for which the path is `"/document"`. The root element is the default context for skills.

Simple paths

Simple paths through the internal enriched document can be expressed with simple tokens separated by slashes. This syntax is similar to [the JSON Pointer specification](#).

Object properties

The properties of nodes that represent objects add their values to the tree under the property's name. Those values can be obtained by appending the property name as a token separated by a slash:

[] [Expand table](#)

Expression	Value
<code>/document/merged_content/language</code>	"en"

Property name tokens are case-sensitive.

Array item index

Specific elements of an array can be referenced by using their numeric index like a property name:

[+] Expand table

Expression	Value
/document/merged_content/keyphrases/1	"Syndrome"
/document/merged_content/entities/0/text	"BMN"

Escape sequences

There are several characters that have a special meaning and need to be escaped if they are to be interpreted as-is instead of a syntax element. These characters include `#`, `/`, and `~` among others.

[+] Expand table

Escape sequence	Special meaning (usage in path syntax)	Example
<code>~0</code>	Used for escaping <code>~</code>	" <code>~0</code> " for <code>~</code> , where " <code>~/documents</code> " becomes " <code>~0~1documents</code> "
<code>~1</code>	Used for escaping <code>/</code>	" <code>~1</code> " for <code>/</code> , where " <code>~/documents</code> " becomes " <code>~0~1documents</code> "
<code>~2</code>	Used for generically to escape arbitrary sequences (including but not limited to <code>#</code> and <code>*</code>)	" <code>~2#~2</code> " where " <code>readme#requirements</code> " becomes " <code>readme~2#~2requirements</code> "

Array enumeration

An array of values can be obtained using the `'*' token`:

[+] Expand table

Expression	Value
/document/normalized_images/0/text/words/*	["Study", "of", "BMN", "110" ...]

The '*' token doesn't have to be at the end of the path. It's possible to enumerate all nodes matching a path with a star in the middle or with multiple stars:

[Expand table](#)

Expression	Value
/document/normalized_images/*/text/words/*	["Study", "of", "BMN", "110" ... "it", "is", "certainly" ...]

This example returns a flat list of all matching nodes.

It's possible to maintain more structure and get a separate array for the words of each page by using a '#' token instead of the second '*' token:

[Expand table](#)

Expression	Value
/document/normalized_images/*/text/words/#	[["Study", "of", "BMN", "110" ...], ["it", "is", "certainly" ...] ...]

The '#' token expresses that the array should be treated as a single value instead of being enumerated.

Enumerating arrays in context

It's often useful to process each element of an array in isolation and have a different set of skill inputs and outputs for each. This can be done by setting the context of the skill to an enumeration instead of the default "/document".

In the following example, we use one of the input expressions we used before, but with a different context that changes the resulting value.

[Expand table](#)

Context	Expression	Values
/document/normalized_images/*	/document/normalized_images/*/text/words/*	["Study", "of", "BMN", "110" ...] ["it", "is", "certainly" ...] ...

For this combination of context and input, the skill gets executed once for each normalized image: once for `"/document/normalized_images/0"` and once for `"/document/normalized_images/1"`. The two input values corresponding to each skill execution are detailed in the values column.

When enumerating an array in context, any outputs the skill produces will also be added to the document as enrichments of the context. In the above example, an output named `"out"` has its values for each execution added to the document respectively under `"/document/normalized_images/0/out"` and `"/document/normalized_images/1/out"`.

Literal values

Skill inputs can take literal values as their inputs instead of dynamic values queried from the existing document. This can be achieved by prefixing the value with an equal sign. Values can be numbers, strings or Boolean. String values can be enclosed in single '`'` or double '`"`' quotes.

 Expand table

Expression	Value
<code>=42</code>	<code>42</code>
<code>=2.45E-4</code>	<code>0.000245</code>
<code>="some string"</code>	<code>"some string"</code>
<code>='some other string'</code>	<code>"some other string"</code>
<code>="unicod\u0065"</code>	<code>"unicode"</code>
<code>=false</code>	<code>false</code>

In line arrays

If a certain skill input requires an array of data, but the data is represented as a single value currently or you need to combine multiple different single values into an array field, then you can create an array value inline as part of a skill input expression by wrapping a comma separated list of expressions in brackets (`[` and `]`). The array value can be a combination of expression paths or literal values as needed. You can also create nested arrays within arrays this way.

 Expand table

Expression	Value
=['item']	["item"]
=[\$(/document/merged_content/entities/0/text), 'item']	["BMN", "item"]
=[1, 3, 5]	[1, 3, 5]
=[true, true, false]	[true, true, false]
=[[\$(/document/merged_content/entities/0/text), 'item'], ['item2', \$(/document/merged_content/keyphrases/1)]]	[["BMN", "item"], ["item2", "Syndrome"]]

If the skill has a context that explains to run the skill per an array input (that is, how "context": "/document/pages/*" means the skill runs once per "page" in `pages`) then passing that value as the expression as input to an in line array uses one of those values at a time.

For an example with our sample enriched data, if your skill's `context` is

`/document/merged_content/keyphrases/*` and then you create an inline array of the following `= ['key phrase', $(/document/merged_content/keyphrases/*)]` on an input of that skill, then the skill is executed three times, once with a value of `["key phrase", "Study of BMN"]`, another with a value of `["key phrase", "Syndrome"]`, and finally with a value of `["key phrase", "Pediatric Patients"]`. The literal "key phrase" value stays the same each time, but the value of the expression path changes with each skill execution.

Composite expressions

It's possible to combine values together using unary, binary, and ternary operators. Operators can combine literal values and values resulting from path evaluation. When used inside an expression, paths should be enclosed between `"$("` and `")"`.

Boolean not `'!'`

[\[\] Expand table](#)

Expression	Value
= <code>!false</code>	true

Negative `'-'`

[\[\] Expand table](#)

Expression	Value
=-42	-42
=-\$(/document/merged_content/entities/0/offset)	-9

Addition '+'

[\[+\] Expand table](#)

Expression	Value
=2+2	4
=2+\$(/document/merged_content/entities/0/offset)	11

Subtraction '-'

[\[+\] Expand table](#)

Expression	Value
=2-1	1
=\$(/document/merged_content/entities/0/offset)-2	7

Multiplication '*'

[\[+\] Expand table](#)

Expression	Value
=2*3	6
=\$(/document/merged_content/entities/0/offset)*2	18

Division '/'

[\[+\] Expand table](#)

Expression	Value
=3/2	1.5

Expression	Value
<code>=\$(/document/merged_content/entities/0/offset)/3</code>	3

Modulo '%'

[+] [Expand table](#)

Expression	Value
<code>=15%4</code>	3
<code>=\$(/document/merged_content/entities/0/offset)%2</code>	1

String concatenation '+'

[+] [Expand table](#)

Expression	Value
<code>="Hello," + "world!"</code>	"Hello, world!"
<code>=\$(/document/merged_content/entities/0/text) + \$(/document/merged_content/entities/0/category)</code>	"BMN Organization"

Less than, less than or equal, greater than and greater than or equal '`<`' '`<=`' '`>`' '`>=`'

[+] [Expand table](#)

Expression	Value
<code>=15<4</code>	false
<code>=4<=4</code>	true
<code>=15>4</code>	true
<code>=1>=2</code>	false

Equality and nonequality '`==`' '`!=`'

[\[+\] Expand table](#)

Expression	Value
=15==4	false
=4==4	true
=15!=4	true
=1!=1	false

Logical operations and, or and exclusive or `'&&'` `'||'` `'^'`

[\[+\] Expand table](#)

Expression	Value
=true&&true	true
=true&&false	false
=true true	true
=true false	true
=false false	false
=true^false	true
=true^true	false

Ternary operator `'?:'`

It's possible to give an input different values based on the evaluation of a Boolean expression using the ternary operator.

[\[+\] Expand table](#)

Expression	Value
=true?"true":"false"	"true"
=\$(/document/merged_content/entities/0/offset)==9?"nine":"not nine"	"nine"

Parentheses and operator priority

Operators are evaluated with priorities that match usual conventions: unary operators, then multiplication, division and modulo, then addition and subtraction, then comparison, then equality, and then logical operators. Usual associativity rules also apply.

Parentheses can be used to change or disambiguate evaluation order.

[+] [Expand table](#)

Expression	Value
=3*2+5	11
=3*(2+5)	21

See also

- [Create a skillset in Azure AI Search](#)
- [Reference enrichments in an Azure AI Search skillset](#)

Azure Vision multimodal embeddings skill

Important

This skill is in public preview under [Supplemental Terms of Use](#). The [2024-05-01-Preview REST API](#) and newer preview APIs support this feature.

The Azure Vision multimodal embeddings skill uses the [multimodal embeddings API](#) from Azure Vision in Foundry Tools to generate embeddings for text or image input.

For transactions that exceed 20 documents per indexer per day, this skill requires that you [attach a billable Microsoft Foundry resource to your skillset](#). Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#). Image extraction is also [billable by Azure AI Search](#).

Location of resources is a consideration for billing. Because you're using a preview REST API version to create a skillset that contains preview skills, you can use a [keyless connection](#) to bypass the same-region requirement. However, for key-based connections, Azure AI Search and Foundry must be in the same region. To ensure region compatibility:

1. Find a [supported region for multimodal embeddings](#).
2. Verify the [region provides AI enrichment](#).

The Foundry resource is used for billing purposes only. Content processing occurs on separate resources managed and maintained by Azure AI Search within the same geo. Your data is processed in the [Geo](#) where your resource is deployed.

@odata.type

Microsoft.Skills.Vision.VectorizeSkill

Data limits

The input limits for the skill can be found in the [Azure Vision documentation](#) for images and text. Consider using the [Text Split skill](#) if you need data chunking for text inputs.

Applicable inputs include:

- Image input file size must be less than 20 megabytes (MB). Image size must be greater than 10 x 10 pixels and less than 16,000 x 16,000 pixels.
- Text input string must be between (inclusive) one word and 70 words.

Skill parameters

Parameters are case sensitive.

[] [Expand table](#)

Inputs	Description
<code>modelVersion</code>	(Required) The model version (<code>2023-04-15</code>) to be passed to the Azure Vision multimodal embeddings API for generating embeddings. Vector embeddings can only be compared and matched if they're from the same model type. Images vectorized by one model won't be searchable through a different model. The latest Image Analysis API offers two models: <ul style="list-style-type: none">• The <code>2023-04-15</code> version, which supports text search in many languages. Azure AI Search uses this version.• The legacy <code>2022-04-11</code> model, which supports only English.

Skill inputs

Skill definition inputs include name, source, and inputs. The following table provides valid values for name of the input. You can also specify recursive inputs. For more information, see the [REST API reference](#) and [Create a skillset](#).

[] [Expand table](#)

Input	Description
<code>text</code>	The input text to be vectorized. If you're using data chunking, the source might be <code>/document/pages/*</code> .
<code>image</code>	Complex Type. Currently only works with <code>/document/normalized_images</code> field, produced by the Azure blob indexer when <code>imageAction</code> is set to a value other than <code>none</code> .
<code>url</code>	The URL to download the image to be vectorized.
<code>queryString</code>	The query string of the URL to download the image to be vectorized. Useful if you store the URL and SAS token in separate paths.

Only one of `text`, `image` or `url/queryString` can be configured for a single instance of the skill. If you want to vectorize both images and text within the same skillset, include two instances of this skill in the skillset definition, one for each input type you would like to use.

Skill outputs

Output	Description
vector	Output embedding array of floats for the input text or image.

Sample definition

For text input, consider a blob that has the following content:

JSON

```
{
    "content": "Forests, grasslands, deserts, and mountains are all part of the
Patagonian landscape that spans more than a million square kilometers of South
America."
}
```

For text inputs, your skill definition might look like this:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
    "context": "/document",
    "modelVersion": "2023-04-15",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "vector",
            "targetName": "text_vector"
        }
    ]
}
```

For image input, a second skill definition in the same skillset might look like this:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
    "context": "/document/normalized_images/*",
    "modelVersion": "2023-04-15",
```

```
"inputs": [
  {
    "name": "image",
    "source": "/document/normalized_images/*"
  }
],
"outputs": [
  {
    "name": "vector",
    "targetName": "image_vector"
  }
]
```

If you want to vectorize images directly from your blob storage data source rather than extract images during indexing, your skill definition should specify a URL, and perhaps a SAS token depending on storage security. For this scenario, your skill definition might look like this:

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Vision.VectorizeSkill",
  "context": "/document",
  "modelVersion": "2023-04-15",
  "inputs": [
    {
      "name": "url",
      "source": "/document/metadata_storage_path"
    },
    {
      "name": "queryString",
      "source": "/document/metadata_storage_sas_token"
    }
  ],
  "outputs": [
    {
      "name": "vector",
      "targetName": "image_vector"
    }
  ]
}
```

Sample output

For the given input, a vectorized embedding output is produced. Output is 1,024 dimensions, which is the number of dimensions supported by the Azure Vision multimodal API.

JSON

```
{  
  "text_vector": [  
    0.018990106880664825,  
    -0.0073809814639389515,  
    ....  
    0.021276434883475304,  
  ]  
}
```

The output resides in memory. To send this output to a field in the search index, you must define an [outputFieldMapping](#) that maps the vectorized embedding output (which is an array) to a [vector field](#). Assuming the skill output resides in the document's **vector** node, and **content_vector** is the field in the search index, the outputFieldMapping in the indexer should look like:

JSON

```
"outputFieldMappings": [  
  {  
    "sourceFieldName": "/document/vector/*",  
    "targetFieldName": "content_vector"  
  }  
]
```

For mapping image embeddings to the index, you use [index projections](#). The payload for `indexProjections` might look something like the following example. `image_content_vector` is a field in the index, and it's populated with the content found in the **vector** of the **normalized_images** array.

JSON

```
"indexProjections": {  
  "selectors": [  
    {  
      "targetIndexName": "myTargetIndex",  
      "parentKeyFieldName": "ParentKey",  
      "sourceContext": "/document/normalized_images/*",  
      "mappings": [  
        {  
          "name": "image_content_vector",  
          "source": "/document/normalized_images/*/vector"  
        }  
      ]  
    }  
  ]  
}
```

See also

- [Built-in skills](#)
 - [How to define a skillset](#)
 - [Extract text and information from images](#)
 - [How to define output fields mappings](#)
 - [Index Projections](#)
 - [Azure Vision multimodal embeddings API](#)
-

Last updated on 11/18/2025

Document Layout skill

The Document Layout skill analyzes a document to detect structure and characteristics, and produces a syntactical representation of the document in Markdown or Text format. You can use it to extract text and images, where image extraction includes location metadata that preserves image position within the document. Image proximity to related content is beneficial in Retrieval Augmented Generation (RAG) workloads and [multimodal search](#) scenarios.

This article is the reference documentation for the Document Layout skill. For usage information, see [How to chunk and vectorize by document layout](#).

This skill uses the [layout model](#) from [Azure Document Intelligence in Foundry Tools](#).

This skill is bound to a [billable Microsoft Foundry resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

💡 Tip

It's common to use this skill on content such as PDFs that have structure and images. The following tutorials demonstrate image verbalization with two different data chunking techniques:

- [Tutorial: Verbalize images from a structured document layout](#)
- [Tutorial: Vectorize from a structured document layout](#)

Limitations

This skill has the following limitations:

- The skill isn't suitable for large documents requiring more than 5 minutes of processing in the Azure Document Intelligence layout model. The skill times out, but charges still apply to the Foundry resource if it's attached to the skillset for billing purposes. Ensure documents are optimized to stay within processing limits to avoid unnecessary costs.
- Since this skill calls the Azure Document Intelligence layout model, all documented [service behaviors for different document types](#) for different file types apply to its output. For example, Word (DOCX) and PDF files may produce different results due to differences in how images are handled. If consistent image behavior across DOCX and PDF is required, consider converting documents to PDF or reviewing the [multimodal search documentation](#) for alternative approaches.

Supported regions

The Document Layout skill calls the [Azure Document Intelligence 2024-11-30 API version V4.0](#).

Supported regions vary by modality and how the skill connects to the Azure Document Intelligence layout model.

 [Expand table](#)

Approach	Requirement
Import data (new) wizard	Create a Foundry resource in one of these regions to get the portal experience: East US, West Europe, North Central US .
Programmatic, using Microsoft Entra ID authentication (preview) for billing	Create Azure AI Search in one of the regions where the service is supported, according to Product availability by region . Create the Foundry resource in any region listed in the Product availability by region table.
Programmatic, using a Foundry resource key for billing	Create your Azure AI Search service and Foundry resource in the same region. This means that the region chosen must have support for both Azure AI Search and Azure Document Intelligence services .

The implemented version of Document Layout model doesn't have support for [21Vianet](#) regions at this time.

Supported file formats

This skill recognizes the following file formats.

- .PDF
- .JPEG
- .JPG
- .PNG
- .BMP
- .TIFF
- .DOCX
- .XLSX
- .PPTX
- .HTML

Supported languages

Refer to [Azure Document Intelligence layout model supported languages](#) for printed text.

@odata.type

Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill

Data limits

- For PDF and TIFF, up to 2,000 pages can be processed (with a free tier subscription, only the first two pages are processed).
- Even if the file size for analyzing documents is 500 MB for [Azure Document Intelligence paid \(S0\) tier](#) and 4 MB for [Azure Document Intelligence free \(F0\) tier](#), indexing is subject to the [indexer limits](#) of your search service tier.
- Image dimensions must be between 50 pixels x 50 pixels or 10,000 pixels x 10,000 pixels.
- If your PDFs are password-locked, remove the lock before running the indexer.

Skill parameters

Parameters are case sensitive. Several parameters were introduced in specific preview versions of the REST API. We recommend using the generally available version (2025-09-01) or the latest preview (2025-11-01-preview) for full access to all parameters.

[] Expand table

Parameter name	Allowed Values	Description
outputMode	oneToMany	Controls the cardinality of the output produced by the skill.
markdownHeaderDepth	h1, h2, h3, h4, h5, h6(default)	Only applies if <code>outputFormat</code> is set to <code>markdown</code> . This parameter describes the deepest nesting level that should be considered. For instance, if the <code>markdownHeaderDepth</code> is <code>h3</code> , any sections that are deeper such as <code>h4</code> , are rolled into <code>h3</code> .
outputFormat	markdown(default), text	New. Controls the format of the output generated by the skill.
extractionOptions	["images"], ["images", "locationMetadata"], ["locationMetadata"]	New. Identify any extra content extracted from the document. Define an array of enums that correspond to the content to be included in the output. For instance, if the <code>extractionOptions</code> is <code>["images", "locationMetadata"]</code> , the output includes images and location metadata which provides page location

Parameter name	Allowed Values	Description
		information related to where the content was extracted, such as a page number or section. This parameter applies to both output formats.
chunkingProperties	See below.	New. Only applies if <code>outputFormat</code> is set to <code>text</code> . Options that encapsulate how to chunk text content while recomputing other metadata.

[+] [Expand table](#)

ChunkingProperties Parameter	Version	Allowed Values	Description
<code>unit</code>	<code>Characters</code> . currently the only allowed value. Chunk length is measured in characters, as opposed to words or tokens	New. Controls the cardinality of the chunk unit.	
<code>maxLength</code>	Any integer between 300-50000	New. The maximum chunk length in characters as measured by <code>String.Length</code> .	
<code>overlapLength</code>	Integer. The value needs to be less than the half of the <code>maxLength</code>	New. The length of overlap provided between two text chunks.	

Skill inputs

[+] [Expand table](#)

Input name	Description
<code>file_data</code>	The file that content should be extracted from.

The "file_data" input must be an object defined as:

JSON

```
{
  "$type": "file",
  "data": "BASE64 encoded string of the file"
}
```

Alternatively, it can be defined as:

JSON

```
{  
  "$type": "file",  
  "url": "URL to download file",  
  "sasToken": "OPTIONAL: SAS token for authentication if the URL provided is for a  
file in blob storage"  
}
```

The file reference object can be generated in one of following ways:

- Setting the `allowSkillsetToReadFileData` parameter on your indexer definition to true. This setting creates a path `/document/file_data` that's an object representing the original file data downloaded from your blob data source. This parameter only applies to files in Azure Blob storage.
- Having a custom skill returning a JSON object definition that provides `$type`, `data`, or `url` and `sastoken`. The `$type` parameter must be set to `file`, and `data` must be the base 64-encoded byte array of the file content. The `url` parameter must be a valid URL with access for downloading the file at that location.

Skill outputs

 Expand table

Output name	Description
<code>markdown_document</code>	Only applies if <code>outputFormat</code> is set to <code>markdown</code> . A collection of "sections" objects, which represent each individual section in the Markdown document.
<code>text_sections</code>	Only applies if <code>outputFormat</code> is set to <code>text</code> . A collection of text chunk objects, which represent the text within the bounds of a page (factoring in any more chunking configured), <i>inclusive</i> of any section headers themselves. The text chunk object includes <code>locationMetadata</code> if applicable.
<code>normalized_images</code>	Only applies if <code>outputFormat</code> is set to <code>text</code> and <code>extractionOptions</code> includes <code>images</code> . A collection of images that were extracted from the document, including <code>locationMetadata</code> if applicable.

Sample definition for markdown output mode

JSON

```
{  
  "skills": [  
    {
```

```
{
  "description": "Analyze a document",
  "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",
  "context": "/document",
  "outputMode": "oneToMany",
  "markdownHeaderDepth": "h3",
  "inputs": [
    {
      "name": "file_data",
      "source": "/document/file_data"
    }
  ],
  "outputs": [
    {
      "name": "markdown_document",
      "targetName": "markdown_document"
    }
  ]
}
```

Sample output for markdown output mode

JSON

```
{
  "markdown_document": [
    {
      "content": "Hi this is Jim \r\nHi this is Joe",
      "sections": {
        "h1": "Foo",
        "h2": "Bar",
        "h3": ""
      },
      "ordinal_position": 0
    },
    {
      "content": "Hi this is Lance",
      "sections": {
        "h1": "Foo",
        "h2": "Bar",
        "h3": "Boo"
      },
      "ordinal_position": 1,
    }
  ]
}
```

The value of the `markdownHeaderDepth` controls the number of keys in the "sections" dictionary. In the example skill definition, since the `markdownHeaderDepth` is "h3," there are three keys in the

"sections" dictionary: h1, h2, h3.

Example for text output mode and image and metadata extraction

This example demonstrates how to output text content in fixed-sized chunks and extract images along with location metadata from the document.

Sample definition for text output mode and image and metadata extraction

JSON

```
{  
  "skills": [  
    {  
      "description": "Analyze a document",  
      "@odata.type": "#Microsoft.Skills.Util.DocumentIntelligenceLayoutSkill",  
      "context": "/document",  
      "outputMode": "oneToMany",  
      "outputFormat": "text",  
      "extractionOptions": ["images", "locationMetadata"],  
      "chunkingProperties": {  
        "unit": "characters",  
        "maxLength": 2000,  
        "overlapLength": 200  
      },  
      "inputs": [  
        {  
          "name": "file_data",  
          "source": "/document/file_data"  
        }  
      ],  
      "outputs": [  
        {  
          "name": "text_sections",  
          "targetName": "text_sections"  
        },  
        {  
          "name": "normalized_images",  
          "targetName": "normalized_images"  
        }  
      ]  
    }  
  ]  
}
```

Sample output for text output mode and image and metadata extraction

JSON

```
"text_sections": [
{
    "id": "1_7e6ef1f0-d2c0-479c-b11c-5d3c0fc88f56",
    "content": "the effects of analyzers using Analyze Text (REST). For more information about analyzers, see Analyzers for text processing. During indexing, an indexer only checks field names and types. There's no validation step that ensures incoming content is correct for the corresponding search field in the index. Create an indexerWhen you're ready to create an indexer on a remote search service, you need a search client. A search client can be the Azure portal, a REST client, or code that instantiates an indexer client. We recommend the Azure portal or REST APIs for early development and proof-of-concept testing. Azure portal1. Sign in to the Azure portal 2, then find your search service.2. On the search service Overview page, choose from two options:· Import data wizard: The wizard is unique in that it creates all of the required elements. Other approaches require a predefined data source and index. All services > Azure AI services | AI Search >demo-search-svc Search serviceSearchAdd indexImport dataImport and vectorize dataOverviewActivity logEssentialsAccess control (IAM)Get startedPropertiesUsageMonitoring· Add indexer: A visual editor for specifying an indexer definition.",

    "locationMetadata": {
        "pageNumber": 1,
        "ordinalPosition": 0,
        "boundingPolygons": "[[{"x":1.5548,"y":0.4036},
{"x":6.9691,"y":0.4033},{("x":6.9691,"y":0.8577},
{"x":1.5548,"y":0.8581}], [{"x":1.181,"y":1.0627},
 {"x":7.1393,"y":1.0626},{("x":7.1393,"y":1.7363}, {"x":1.181,"y":1.7365}],
 [{"x":1.1923,"y":2.1466},{("x":3.4585,"y":2.1496},
 {"x":3.4582,"y":2.4251}, {"x":1.1919,"y":2.4221}],
 [{"x":1.1813,"y":2.6518},{("x":7.2464,"y":2.6375},
 {"x":7.2486,"y":3.5913}, {"x":1.1835,"y":3.6056}],
 [{"x":1.3349,"y":3.9489}, {"x":2.1237,"y":3.9508},
 {"x":2.1233,"y":4.1128}, {"x":1.3346,"y":4.111}],
 [{"x":1.5705,"y":4.5322}, {"x":5.801,"y":4.5326}, {"x":5.801,"y":4.7311},
 {"x":1.5704,"y":4.7307}]]"
    },
    "sections": []
},
{
    "id": "2_25134f52-04c3-415a-ab3d-80729bd58e67",
    "content": "All services > Azure AI services | AI Search >demo-search-svc | Indexers Search serviceSearch0«Add indexerRefreshDelete:selected: TagsFilter by name ...:selected: Diagnose and solve problemsSearch managementStatusNameIndexesIndexers*Data sourcesRun the indexerBy default, an indexer runs immediately when you create it on the search service. You can override this behavior by setting disabled to true in the indexer definition. Indexer execution is the moment of truth where you find out if there are problems with connections, field mappings, or skillset construction. There are several ways to run an indexer:: Run on indexer creation or update (default).. Run on demand when there

```

are no changes to the definition, or precede with `reset` for full indexing. For more information, see Run or reset indexers.. Schedule indexer processing to invoke execution at regular intervals.Scheduled execution is usually implemented when you have a need for incremental indexing so that you can pick up the latest changes. As such, scheduling has a dependency on change detection.Indexers are one of the few subsystems that make overt outbound calls to other Azure resources. In terms of Azure roles, indexers don't have separate identities; a connection from the search engine to another Azure resource is made using the system or user- assigned managed identity of a search service. If the indexer connects to an Azure resource on a virtual network, you should create a shared private link for that connection. For more information about secure connections, see Security in Azure AI Search.Check results",

```
    "locationMetadata": {
        "pageNumber": 2,
        "ordinalPosition": 1,
        "boundingPolygons": "[[{"x":2.2041,"y":0.4109},
{"x":4.3967,"y":0.4131},{x":4.3966,"y":0.5505},{x":2.204,"y":0.5482}], [{"x":2.5042,"y":0.6422},{x":4.8539,"y":0.6506},{x":4.8527,"y":0.993},{x":2.5029,"y":0.9845}],[{x":2.3705,"y":1.1496},{x":2.6859,"y":1.15},{x":2.6858,"y":1.2612},{x":2.3704,"y":1.2608}], [{"x":3.7418,"y":1.1709},{x":3.8082,"y":1.171},{x":3.8081,"y":1.2508},{x":3.7417,"y":1.2507}],[{x":3.9692,"y":1.1445},{x":4.0541,"y":1.1445},{x":4.0542,"y":1.2621},{x":3.9692,"y":1.2622}],[{x":4.5326,"y":1.2263},{x":5.1065,"y":1.229},{x":5.106,"y":1.346},{x":4.5321,"y":1.3433}],[{x":5.5508,"y":1.2267},{x":5.8992,"y":1.2268},{x":5.8991,"y":1.3408},{x":5.5508,"y":1.3408}]]"
    },
    "sections": []
}
],
"normalized_images": [
{
    "id": "1_550e8400-e29b-41d4-a716-446655440000",
    "data": "SGVsbG8sIFdvcmxkIQ==",
    "imagePath": "aHR0cHM6Ly9henNyb2xsaw5nLmJsb2IuY29yZS53aw5kb3dzLm5ldC9tdWx0aW1vZGFsaXR5L0NyZWF0ZU1uZGV4ZXJwNnA3LnBkZg2/normalized_images_0.jpg",
    "locationMetadata": {
        "pageNumber": 1,
        "ordinalPosition": 0,
        "boundingPolygons": "[[{"x":2.0834,"y":6.2245},
{"x":7.1818,"y":6.2244},{x":7.1816,"y":7.9375},{x":2.0831,"y":7.9377}]]"
    }
},
{
    "id": "2_123e4567-e89b-12d3-a456-426614174000",
    "data": "U29tZSBtb3JlIGV4YW1wbGUgdGV4dA==",
    "imagePath": "aHR0cHM6Ly9henNyb2xsaw5nLmJsb2IuY29yZS53aw5kb3dzLm5ldC9tdWx0aW1vZGFsaXR5L0NyZWF0ZU1uZGV4ZXJwNnA3LnBkZg2/normalized_images_1.jpg",
    "locationMetadata": {
        "pageNumber": 2,
        "ordinalPosition": 1,
    }
}
```

```
        "boundingPolygons": "[[{"x":2.0784,"y":0.3734},  
{"x":7.1837,"y":0.3729}, {"x":7.183,"y":2.8611},  
 {"x":2.0775,"y":2.8615}]]"  
    }  
}  
]
```

Note that the “sections” in the sample output above appear blank. To populate them, you’ll need to add an additional skill configured with `outputFormat` set to `markdown` to ensure the sections are properly filled.

The skill uses [Azure Document Intelligence](#) to compute locationMetadata. Refer to [Azure Document Intelligence layout model](#) for details on how pages and bounding polygon coordinates are defined.

The `imagePath` represents the relative path of a stored image. If the knowledge store file projection is configured in the skillset, this path matches the relative path of the image stored in the knowledge store.

See also

- [What is the Azure Document Intelligence layout model](#)
- [Built-in skills](#)
- [How to define a skill set](#)
- [Create Indexer \(REST API\)](#)

Last updated on 11/19/2025

Entity Linking cognitive skill (v3)

The Entity Linking skill (v3) returns a list of recognized entities with links to articles in a well-known knowledge base (Wikipedia).

! Note

This skill is bound to the [Entity Linking](#) machine learning models in [Azure Vision in Foundry Tools](#). It requires a [billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

@odata.type

Microsoft.Skills.Text.V3.EntityLinkingSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the EntityLinking skill, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameter names are case-sensitive and are all optional.

[+] [Expand table](#)

Parameter name	Description
<code>defaultLanguageCode</code>	Language code of the input text. If the default language code isn't specified, English (en) is used as the default language code. See the full list of supported languages .
<code>minimumPrecision</code>	A value between 0 and 1. If the confidence score (in the <code>entities</code> output) is lower than this value, the entity isn't returned. The default is 0.
<code>modelVersion</code>	(Optional) Specifies the version of the model to use when calling entity linking. It defaults to the latest available when not specified. We recommend you don't specify this value unless it's necessary.

Skill inputs

[\[\] Expand table](#)

Input name	Description
languageCode	A string indicating the language of the records. If this parameter isn't specified, the default language code is used to analyze the records. See the full list of supported languages .
text	The text to analyze.

Skill outputs

[\[\] Expand table](#)

Output	Description
name	
entities	An array of complex types that contains the following fields: <ul style="list-style-type: none">• "name" (The actual entity name as it appears in the text)• "id"• "language" (The language of the text as determined by the skill)• "url" (The linked url to this entity)• "bingId" (The bingId for this linked entity)• "dataSource" (The data source associated with the url)• "matches" (An array of complex types that contains: <code>text</code>, <code>offset</code>, <code>length</code> and <code>confidenceScore</code>)

Sample definition

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityLinkingSkill",  
  "context": "/document",  
  "defaultLanguageCode": "en",  
  "minimumPrecision": 0.5,  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content"  
    },  
    {  
      "name": "languageCode",  
      "source": "/document/languageCode"  
    }  
  ]}
```

```
        "source": "/document/language"
    }
],
"outputs": [
{
    "name": "entities",
    "targetName": "entities"
}
]
}
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data":
            {
                "text": "Microsoft is liked by many.",
                "languageCode": "en"
            }
        }
    ]
}
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data" :
            {
                "entities": [
                    {
                        "name": "Microsoft",
                        "id": "Microsoft",
                        "language": "en",
                        "url": "https://en.wikipedia.org/wiki/Microsoft",
                        "bingId": "a093e9b9-90f5-a3d5-c4b8-5855e1b01f85",
                        "dataSource": "Wikipedia",
                        "matches": [
                            {
                                "text": "Microsoft",

```

```
        "offset": 0,
        "length": 9,
        "confidenceScore": 0.13
    }
]
}
],
}
]
}
```

The offsets returned for entities in the output of this skill are directly returned from the [Language Service APIs](#), which means if you're using them to index into the original string, you should use the [StringInfo](#) class in .NET in order to extract the correct content. For more information, see [Multilingual and emoji support in Language service features](#).

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Last updated on 11/18/2025

Entity Recognition cognitive skill (v3)

The Entity Recognition skill (v3) extracts entities of different types from text. These entities fall under 14 distinct categories, ranging from people and organizations to URLs and phone numbers. This skill uses the [Named Entity Recognition](#) machine learning models provided by [Azure Language in Foundry Tools](#).

! Note

This skill is bound to Foundry Tools and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

@odata.type

Microsoft.Skills.Text.V3.EntityRecognitionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the EntityRecognition skill, consider using the [Text Split skill](#). When using a split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case sensitive and are all optional.

[] Expand table

Parameter name	Description
categories	Array of categories that should be extracted. Possible category types: "Person", "Location", "Organization", "Quantity", "DateTime", "URL", "Email", "personType", "Event", "Product", "Skill", "Address", "phoneNumber", "ipAddress". If no category is provided, all types are returned.
defaultLanguageCode	Language code of the input text. If the default language code is not specified, English (en) will be used as the default language code. See the full list of supported languages . Not all entity categories are supported for all languages; see note below.

Parameter name	Description
<code>minimumPrecision</code>	A value between 0 and 1. If the confidence score (in the <code>namedEntities</code> output) is lower than this value, the entity is not returned. The default is 0.
<code>modelVersion</code>	(Optional) Specifies the version of the model to use when calling the entity recognition API. It will default to the latest available when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

[] [Expand table](#)

Input name	Description
<code>languageCode</code>	A string indicating the language of the records. If this parameter is not specified, the default language code will be used to analyze the records. See the full list of supported languages .
<code>text</code>	The text to analyze.

Skill outputs

! Note

Not all entity categories are supported for all languages. See [Supported Named Entity Recognition \(NER\) entity categories](#) to know which entity categories are supported for the language you will be using.

[] [Expand table](#)

Output name	Description
<code>persons</code>	An array of strings where each string represents the name of a person.
<code>locations</code>	An array of strings where each string represents a location.
<code>organizations</code>	An array of strings where each string represents an organization.
<code>quantities</code>	An array of strings where each string represents a quantity.
<code>dateTimes</code>	An array of strings where each string represents a DateTime (as it appears in the text) value.

Output name	Description
urls	An array of strings where each string represents a URL
emails	An array of strings where each string represents an email
personTypes	An array of strings where each string represents a PersonType
events	An array of strings where each string represents an event
products	An array of strings where each string represents a product
skills	An array of strings where each string represents a skill
addresses	An array of strings where each string represents an address
phoneNumbers	An array of strings where each string represents a telephone number
ipAddresses	An array of strings where each string represents an IP Address
namedEntities	An array of complex types that contains the following fields: <ul style="list-style-type: none"> category subcategory confidenceScore (Higher value means it's more likely to be a real entity) length (The length(number of characters) of this entity) offset (The location where it was found in the text) text (The actual entity name as it appears in the text)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "context": "/document",
  "categories": [ "Person", "Email" ],
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.5,
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    },
    {
      "name": "languageCode",
      "source": "/document/language"
    }
  ],
  "outputs": [
    {
      "name": "entities"
    }
  ]
}
```

```
        "name": "persons",
        "targetName": "people"
    },
    {
        "name": "emails",
        "targetName": "emails"
    },
    {
        "name": "namedEntities",
        "targetName": "namedEntities"
    }
]
}
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data":
                {
                    "text": "Contoso Corporation was founded by Jean Martin. They can be reached at contact@contoso.com",
                    "languageCode": "en"
                }
        }
    ]
}
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data" :
                {
                    "people": [ "Jean Martin"],
                    "emails": ["contact@contoso.com"],
                    "namedEntities":
                        [
                            {
                                "category": "Person",
                                "subcategory": null,

```

```
        "length": 11,
        "offset": 35,
        "confidenceScore": 0.98,
        "text": "Jean Martin"
    },
    {
        "category": "Email",
        "subcategory": null,
        "length": 19,
        "offset": 71,
        "confidenceScore": 0.8,
        "text": "contact@contoso.com"
    }
],
}
}
]
}
```

The offsets returned for entities in the output of this skill are directly returned from the [Language Service APIs](#), which means if you are using them to index into the original string, you should use the [StringInfo](#) class in .NET in order to extract the correct content. For more information, see [Multilingual and emoji support in Language service features](#).

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Image Analysis cognitive skill

The **Image Analysis** skill extracts a rich set of visual features based on the image content. For example, you can generate a caption from an image, generate tags, or identify celebrities and landmarks. This article is the reference documentation for the **Image Analysis** skill. See [Extract text and information from images](#) for usage instructions.

This skill uses the machine learning models provided by [Azure Vision in Foundry Tools](#). **Image Analysis** works on images that meet the following requirements:

- The image must be presented in JPEG, PNG, GIF or BMP format
- The file size of the image must be less than 4 megabytes (MB)
- The dimensions of the image must be greater than 50 x 50 pixels

Supported data sources for OCR and image analysis are blobs in Azure Blob Storage and Azure Data Lake Storage (ADLS) Gen2, and image content in Microsoft OneLake. Images can be standalone files or embedded images in a PDF or other files.

This skill is implemented using the [AI Image Analysis API](#) version 3.2. If your solution requires calling a newer version of that service API (such as version 4.0), consider implementing through [Web API custom skill](#) or use the [ImageAnalysisV4 power skill](#).

(!) Note

This skill is bound to Foundry Tools and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

In addition, image extraction is [billable by Azure AI Search](#).

@odata.type

Microsoft.Skills.Vision.ImageAnalysisSkill

Skill parameters

Parameters are case sensitive.

[+] [Expand table](#)

Parameter name	Description
<code>defaultLanguageCode</code>	<p>A string indicating the language to return. The service returns recognition results in a specified language. If this parameter isn't specified, the default value is "en".</p> <p>Supported languages include a subset of generally available languages of Azure Vision. When a language is newly introduced with general availability status into Azure Vision, there is expected delay before they are fully integrated within this skill.</p>
<code>visualFeatures</code>	<p>An array of strings indicating the visual feature types to return. Valid visual feature types include:</p> <ul style="list-style-type: none"> • <i>adult</i> - detects if the image is pornographic (depicts nudity or a sex act), gory (depicts extreme violence or blood) or suggestive (also known as racy content). • <i>brands</i> - detects various brands within an image, including the approximate location. • <i>categories</i> - categorizes image content according to a taxonomy defined by Foundry Tools. • <i>description</i> - describes the image content with a complete sentence in supported languages. • <i>faces</i> - detects if faces are present. If present, generates coordinates, gender and age. • <i>objects</i> - detects various objects within an image, including the approximate location. • <i>tags</i> - tags the image with a detailed list of words related to the image content. <p>Names of visual features are case-sensitive. Both <i>color</i> and <i>imageType</i> visual features have been deprecated, but you can access this functionality through a custom skill. Refer to the Azure Vision Image Analysis documentation on which visual features are supported with each <code>defaultLanguageCode</code>.</p>
<code>details</code>	<p>An array of strings indicating which domain-specific details to return. Valid visual feature types include:</p> <ul style="list-style-type: none"> • <i>celebrities</i> - identifies celebrities if detected in the image. • <i>landmarks</i> - identifies landmarks if detected in the image.

Skill inputs

 Expand table

Input name	Description
<code>image</code>	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure blob indexer when <code>imageAction</code> is set to a value other than <code>none</code> .

Skill outputs

[Expand table](#)

Output	Description
<code>name</code>	
<code>adult</code>	Output is a single <code>adult</code> object of a complex type, consisting of boolean fields (<code>isAdultContent</code> , <code>isGoryContent</code> , <code>isRacyContent</code>) and double type scores (<code>adultScore</code> , <code>goreScore</code> , <code>racyScore</code>).
<code>brands</code>	Output is an array of <code>brand</code> objects, where the object is a complex type consisting of <code>name</code> (string) and a <code>confidence</code> score (double). It also returns a <code>rectangle</code> with four bounding box coordinates (<code>x</code> , <code>y</code> , <code>w</code> , <code>h</code> , in pixels) indicating placement inside the image. For the rectangle, <code>x</code> and <code>y</code> are the top left. Bottom left is <code>x</code> , <code>y+h</code> . Top right is <code>x+w</code> , <code>y</code> . Bottom right is <code>x+w</code> , <code>y+h</code> .
<code>categories</code>	Output is an array of <code>category</code> objects, where each category object is a complex type consisting of a <code>name</code> (string), <code>score</code> (double), and optional <code>detail</code> that contains celebrity or landmark details. See the category taxonomy for the full list of category names. A detail is a nested complex type. A celebrity detail consists of a name, confidence score, and face bounding box. A landmark detail consists of a name and confidence score.
<code>description</code>	Output is a single <code>description</code> object of a complex type, consisting of lists of <code>tags</code> and <code>caption</code> (an array consisting of <code>Text</code> (string) and <code>confidence</code> (double)).
<code>faces</code>	Complex type consisting of <code>age</code> , <code>gender</code> , and <code>faceBoundingBox</code> having four bounding box coordinates (in pixels) indicating placement inside the image. Coordinates are <code>top</code> , <code>left</code> , <code>width</code> , <code>height</code> .
<code>objects</code>	Output is an array of visual feature objects . Each object is a complex type, consisting of <code>object</code> (string), <code>confidence</code> (double), <code>rectangle</code> (with four bounding box coordinates indicating placement inside the image), and a <code>parent</code> that contains an object name and confidence .
<code>tags</code>	Output is an array of <code>imageTag</code> objects, where a tag object is a complex type consisting of <code>name</code> (string), <code>hint</code> (string), and <code>confidence</code> (double). The addition of a hint is rare. It's only generated if a tag is ambiguous. For example, an image tagged as "curling" might have a hint of "sports" to better indicate its content.

Sample skill definition

JSON

```
{  
  "description": "Extract image analysis.",  
  "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
```

```
"context": "/document/normalized_images/*",
"defaultLanguageCode": "en",
"visualFeatures": [
    "adult",
    "brands",
    "categories",
    "description",
    "faces",
    "objects",
    "tags"
],
"inputs": [
    {
        "name": "image",
        "source": "/document/normalized_images/*"
    }
],
"outputs": [
    {
        "name": "adult"
    },
    {
        "name": "brands"
    },
    {
        "name": "categories"
    },
    {
        "name": "description"
    },
    {
        "name": "faces"
    },
    {
        "name": "objects"
    },
    {
        "name": "tags"
    }
]
}
```

Sample index

For single objects (such as `adult` and `description`), you can structure them in the index as a `Collection(Edm.ComplexType)` to return `adult` and `description` output for all of them. For more information about mapping outputs to index fields, see [Flattening information from complex types](#).

JSON

```
{
  "fields": [
    {
      "name": "metadata_storage_name",
      "type": "Edm.String",
      "key": true,
      "searchable": true,
      "filterable": false,
      "facetable": false,
      "sortable": true
    },
    {
      "name": "metadata_storage_path",
      "type": "Edm.String",
      "searchable": true,
      "filterable": false,
      "facetable": false,
      "sortable": true
    },
    {
      "name": "content",
      "type": "Edm.String",
      "sortable": false,
      "searchable": true,
      "filterable": false,
      "facetable": false
    },
    {
      "name": "adult",
      "type": "Edm.ComplexType",
      "fields": [
        {
          "name": "isAdultContent",
          "type": "Edm.Boolean",
          "searchable": false,
          "filterable": true,
          "facetable": true
        },
        {
          "name": "isGoryContent",
          "type": "Edm.Boolean",
          "searchable": false,
          "filterable": true,
          "facetable": true
        },
        {
          "name": "isRacyContent",
          "type": "Edm.Boolean",
          "searchable": false,
          "filterable": true,
          "facetable": true
        },
        {
          "name": "adultScore",
          "type": "Edm.Int32"
        }
      ]
    }
  ]
}
```

```
        "type": "Edm.Double",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "goreScore",
        "type": "Edm.Double",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "racyScore",
        "type": "Edm.Double",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
},
{
    "name": "brands",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "name",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "confidence",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "rectangle",
            "type": "Edm.ComplexType",
            "fields": [
                {
                    "name": "x",
                    "type": "Edm.Int32",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                },
                {
                    "name": "y",
                    "type": "Edm.Int32",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                }
            ]
        }
    ]
}
```

```
        "facetable": false
    },
    {
        "name": "w",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "h",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
}
],
{
    "name": "categories",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "name",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "score",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        }
    ],
    {
        "name": "detail",
        "type": "Edm.ComplexType",
        "fields": [
            {
                "name": "celebrities",
                "type": "Collection(Edm.ComplexType)",
                "fields": [
                    {
                        "name": "name",
                        "type": "Edm.String",
                        "searchable": true,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "faceBoundingBox",
                        "type": "Edm.String",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    }
                ]
            }
        ]
    }
}
```

```
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "x",
                "type": "Edm.Int32",
                "searchable": false,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "y",
                "type": "Edm.Int32",
                "searchable": false,
                "filterable": false,
                "facetable": false
            }
        ]
    },
    {
        "name": "confidence",
        "type": "Edm.Double",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
},
{
    "name": "landmarks",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "name",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "confidence",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        }
    ]
}
],
{
    "name": "description",
    "type": "Collection(Edm.ComplexType)",
    "fields": [

```

```
{
    "name": "tags",
    "type": "Collection(Edm.String)",
    "searchable": true,
    "filterable": false,
    "facetable": false
},
{
    "name": "captions",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "text",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "confidence",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        }
    ]
},
{
    "name": "faces",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "age",
            "type": "Edm.Int32",
            "searchable": false,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "gender",
            "type": "Edm.String",
            "searchable": false,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "faceBoundingBox",
            "type": "Collection(Edm.ComplexType)",
            "fields": [
                {
                    "name": "top",
                    "type": "Edm.Int32",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                }
            ]
        }
    ]
}
```

```
        "filterable": false,
        "facetable": false
    },
    {
        "name": "left",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "width",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "height",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
}
],
{
    "name": "objects",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "object",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "confidence",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "rectangle",
            "type": "Edm.ComplexType",
            "fields": [
                {
                    "name": "x",
                    "type": "Edm.Int32",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                }
            ]
        }
    ]
}
```

```
        "facetable": false
    },
    {
        "name": "y",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "w",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "h",
        "type": "Edm.Int32",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
},
{
    "name": "parent",
    "type": "Edm.ComplexType",
    "fields": [
        {
            "name": "object",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "confidence",
            "type": "Edm.Double",
            "searchable": false,
            "filterable": false,
            "facetable": false
        }
    ]
}
],
{
    "name": "tags",
    "type": "Collection(Edm.ComplexType)",
    "fields": [
        {
            "name": "name",
            "type": "Edm.String",
            "searchable": true,
```

```

        "filterable": false,
        "facetable": false
    },
    {
        "name": "hint",
        "type": "Edm.String",
        "searchable": true,
        "filterable": false,
        "facetable": false
    },
    {
        "name": "confidence",
        "type": "Edm.Double",
        "searchable": false,
        "filterable": false,
        "facetable": false
    }
]
}
]
}

```

Sample output field mapping

The target field can be a complex field or collection. The index definition specifies any subfields.

JSON

```

"outputFieldMappings": [
    {
        "sourceFieldName": "/document/normalized_images/*/adult",
        "targetFieldName": "adult"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/brands/*",
        "targetFieldName": "brands"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/categories/*",
        "targetFieldName": "categories"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/description",
        "targetFieldName": "description"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/faces/*",
        "targetFieldName": "faces"
    },
    {

```

```

        "sourceFieldName": "/document/normalized_images/*/objects/*",
        "targetFieldName": "objects"
    },
{
    "sourceFieldName": "/document/normalized_images/*/tags/*",
    "targetFieldName": "tags"
}

```

Variation on output field mappings (nested properties)

You can define output field mappings to lower-level properties, such as just celebrities or landmarks. In this case, make sure your index schema has a field to contain each detail specifically.

JSON

```

"outputFieldMappings": [
    {
        "sourceFieldName":
        "/document/normalized_images/*/categories/detail/celebrities/*",
        "targetFieldName": "celebrities"
    },
    {
        "sourceFieldName":
        "/document/normalized_images/*/categories/detail/landmarks/*",
        "targetFieldName": "landmarks"
    }
]

```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "image": {
                    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",
                    "width": 500,
                    "height": 300,
                    "originalWidth": 5000,
                    "originalHeight": 3000,
                    "rotationFromOriginal": 90,
                    "contentOffset": 500,
                    "pageNumber": 2
                }
            }
        }
    ]
}
```

```
]  
}
```

Sample output

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data": {  
        "categories": [  
          {  
            "name": "abstract_",  
            "score": 0.00390625  
          },  
          {  
            "name": "people_",  
            "score": 0.83984375,  
            "detail": {  
              "celebrities": [  
                {  
                  "name": "Satya Nadella",  
                  "faceBoundingBox": [  
                    {  
                      "x": 273,  
                      "y": 309  
                    },  
                    {  
                      "x": 395,  
                      "y": 309  
                    },  
                    {  
                      "x": 395,  
                      "y": 431  
                    },  
                    {  
                      "x": 273,  
                      "y": 431  
                    }  
                  ],  
                  "confidence": 0.99902844  
                }  
              ]  
            }  
          }  
        ]  
      }  
    ],  
    "adult": {  
      "isAdultContent": false,  
      "isRacyContent": false,  
      "isExplicitContent": false  
    }  
  ]  
}
```

```
"isGoryContent": false,
"adultScore": 0.0934349000453949,
"racyScore": 0.068613491952419281,
"goreScore": 0.08928389008070282
},
"tags": [
{
  "name": "person",
  "confidence": 0.98979085683822632
},
{
  {
    "name": "man",
    "confidence": 0.94493889808654785
  },
  {
    "name": "outdoor",
    "confidence": 0.938492476940155
  },
  {
    "name": "window",
    "confidence": 0.89513939619064331
  }
],
"description": {
  "tags": [
    "person",
    "man",
    "outdoor",
    "window",
    "glasses"
  ],
  "captions": [
    {
      "text": "Satya Nadella sitting on a bench",
      "confidence": 0.48293603002174407
    }
  ]
},
"faces": [
{
  "age": 44,
  "gender": "Male",
  "faceBoundingBox": [
    {
      "x": 1601,
      "y": 395
    },
    {
      "x": 1653,
      "y": 395
    },
    {
      "x": 1653,
      "y": 447
    }
  ]
}
]
```

```

        "x": 1601,
        "y": 447
    }
]
}
],
"objects": [
{
    "rectangle": {
        "x": 25,
        "y": 43,
        "w": 172,
        "h": 140
    },
    "object": "person",
    "confidence": 0.931
}
],
"brands": [
{
    "name": "Microsoft",
    "confidence": 0.903,
    "rectangle": {
        "x": 20,
        "y": 97,
        "w": 62,
        "h": 52
    }
}
]
}
]
}

```

Error cases

In the following error cases, no elements are extracted.

 [Expand table](#)

Error Code	Description
NotSupportedLanguage	The language provided isn't supported.
InvalidImageUrl	Image URL is badly formatted or not accessible.
InvalidImageFormat	Input data isn't a valid image.
InvalidImageSize	Input image is too large.

Error Code	Description
NotSupportedVisualFeature	Specified feature type isn't valid.
NotSupportedImage	Unsupported image, for example, child pornography.
InvalidDetails	Unsupported domain-specific model.

If you get the error similar to "One or more skills are invalid. Details: Error in skill # <num>: Outputs are not supported by skill: Landmarks", check the path. Both celebrities and landmarks are properties under `detail`.

JSON

```
"categories": [
  {
    "name": "building_",
    "score": 0.97265625,
    "detail": {
      "landmarks": [
        {
          "name": "Forbidden City",
          "confidence": 0.92013400793075562
        }
      ]
    }
  }
]
```

See also

- [What is Image Analysis?](#)
- [Built-in skills](#)
- [How to define a skillset](#)
- [Extract text and information from images](#)
- [Create Indexer \(REST\)](#)

Last updated on 11/18/2025

Key Phrase Extraction cognitive skill

The **Key Phrase Extraction** skill evaluates unstructured text, and for each record, returns a list of key phrases. This skill uses the [Key Phrase](#) machine learning models provided by [Azure Language in Foundry Tools](#).

This capability is useful if you need to quickly identify the main talking points in the record. For example, given input text "The food was delicious and there were wonderful staff", the service returns "food" and "wonderful staff".

! Note

This skill is bound to Foundry Tools and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

@odata.type

Microsoft.Skills.Text.KeyPhraseExtractionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the key phrase extractor, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case sensitive.

[\[\] Expand table](#)

Inputs	Description
<code>defaultLanguageCode</code>	(Optional) The language code to apply to documents that don't specify language explicitly. If the default language code isn't specified, English (en) is used as the default language code. See the full list of supported languages .
<code>maxKeyPhraseCount</code>	(Optional) The maximum number of key phrases to produce.

Inputs	Description
<code>modelVersion</code>	(Optional) Specifies the version of the model to use when calling the key phrase API. It defaults to the latest available when not specified. We recommend you don't specify this value unless it's necessary.

Skill inputs

[+] [Expand table](#)

Input	Description
<code>text</code>	The text to be analyzed.
<code>languageCode</code>	A string indicating the language of the records. If this parameter isn't specified, the default language code is used to analyze the records. See the full list of supported languages .

Skill outputs

[+] [Expand table](#)

Output	Description
<code>keyPhrases</code>	A list of key phrases extracted from the input text. The key phrases are returned in order of importance.

Sample definition

Consider a SQL record that has the following fields:

JSON
<pre>{ "content": "Glaciers are huge rivers of ice that ooze their way over land, powered by gravity and their own sheer weight. They accumulate ice from snowfall and lose it through melting. As global temperatures have risen, many of the world's glaciers have already started to shrink and retreat. Continued warming could see many iconic landscapes - from the Canadian Rockies to the Mount Everest region of the Himalayas - lose almost all their glaciers by the end of the century.", "language": "en" }</pre>

Then your skill definition might look like this:

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.KeyPhraseExtractionSkill",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        },  
        {  
            "name": "languageCode",  
            "source": "/document/language"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "keyPhrases",  
            "targetName": "myKeyPhrases"  
        }  
    ]  
}
```

Sample output

For the previous example, the output of your skill is written to a new node in the enriched tree called "document/myKeyPhrases" since that is the `targetName` that we specified. If you don't specify a `targetName`, then it would be "document/keyPhrases".

document/myKeyPhrases

JSON

```
[  
    "world's glaciers",  
    "huge rivers of ice",  
    "Canadian Rockies",  
    "iconic landscapes",  
    "Mount Everest region",  
    "Continued warming"  
]
```

You can use "document/myKeyPhrases" as input into other skills, or as a source of an [output field mapping](#).

Warnings

If you provide an unsupported language code, a warning is generated and key phrases aren't extracted. If your text is empty, a warning is produced. If your text is larger than 50,000 characters, only the first 50,000 characters are analyzed and a warning is issued.

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [How to define output fields mappings](#)

Last updated on 11/18/2025

Language detection cognitive skill

The **Language Detection** skill detects the language of input text and reports a single language code for every document submitted on the request. The language code is paired with a score indicating the strength of the analysis. This skill uses the machine learning models provided in [Azure AI Language](#).

This capability is especially useful when you need to provide the language of the text as input to other skills (for example, the [Sentiment Analysis skill](#) or [Text Split skill](#)).

See [supported languages](#) for Language Detection. If you have content expressed in an unsupported language, the response is `(Unknown)`.

! Note

This skill is bound to Azure AI services and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Azure AI services Standard price](#).

@odata.type

Microsoft.Skills.Text.LanguageDetectionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the language detection skill, you can use the [Text Split skill](#).

Skill parameters

Parameters are case-sensitive.

[+] [Expand table](#)

Inputs	Description
<code>defaultCountryHint</code>	(Optional) An ISO 3166-1 alpha-2 two letter country code can be provided to use as a hint to the language detection model if it can't disambiguate the language . Specifically, the <code>defaultCountryHint</code> parameter is used with documents that don't specify the <code>countryHint</code> input explicitly.

Inputs	Description
<code>modelVersion</code>	(Optional) Specifies the version of the model to use when calling language detection. It defaults to the latest available when not specified. We recommend you don't specify this value unless it's necessary.

Skill inputs

Parameters are case-sensitive.

[\[+\] Expand table](#)

Inputs	Description
<code>text</code>	The text to be analyzed.
<code>countryHint</code>	An ISO 3166-1 alpha-2 two letter country code to use as a hint to the language detection model if it can't disambiguate the language .

Skill outputs

[\[+\] Expand table](#)

Output	Description
Name	
<code>languageCode</code>	The ISO 6391 language code for the language identified. For example, "en".
<code>languageName</code>	The name of language. For example, "English".
<code>score</code>	A value between 0 and 1. The likelihood that language is correctly identified. The score can be lower than 1 if the sentence has mixed languages.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
  "inputs": [
    {
      "name": "text",
      "source": "/document/text"
    },
    {
      "name": "countryHint",
      "source": "/document/countryHint"
    }
  ],
  "outputs": [
    {
      "name": "languageCode",
      "target": "/document/languageCode"
    },
    {
      "name": "languageName",
      "target": "/document/languageName"
    },
    {
      "name": "score",
      "target": "/document/score"
    }
  ]
}
```

```

        "name": "countryHint",
        "source": "/document/countryHint"
    }
],
"outputs": [
{
    "name": "languageCode",
    "targetName": "myLanguageCode"
},
{
    "name": "languageName",
    "targetName": "myLanguageName"
},
{
    "name": "score",
    "targetName": "myLanguageScore"
}
]
}

```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "Glaciers are huge rivers of ice that ooze their way over land, powered by gravity and their own sheer weight. "
      }
    },
    {
      "recordId": "2",
      "data": {
        "text": "Estamos muy felices de estar con ustedes."
      }
    },
    {
      "recordId": "3",
      "data": {
        "text": "impossible",
        "countryHint": "fr"
      }
    }
  ]
}
```

Sample output

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "languageCode": "en",  
          "languageName": "English",  
          "score": 1,  
        }  
    },  
    {  
      "recordId": "2",  
      "data":  
        {  
          "languageCode": "es",  
          "languageName": "Spanish",  
          "score": 1,  
        }  
    },  
    {  
      "recordId": "3",  
      "data":  
        {  
          "languageCode": "fr",  
          "languageName": "French",  
          "score": 1,  
        }  
    }  
  ]  
}
```

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Last updated on 05/19/2025

OCR cognitive skill

The **optical character recognition (OCR)** skill recognizes printed and handwritten text in image files. This article is the reference documentation for the OCR skill. See [Extract text from images](#) for usage instructions.

The **OCR** skill uses the machine learning models provided by [Azure Vision in Foundry Tools API v3.2](#). The **OCR** skill maps to the following functionality:

- For the languages listed under [Azure Vision language support](#), the [Read API](#) is used.
- For Greek and Serbian Cyrillic, the legacy [OCR in version 3.2](#) API is used.

The **OCR** skill extracts text from image files and embedded images. Supported file formats include:

- .JPEG
- .JPG
- .PNG
- .BMP
- .TIFF

Supported data sources for OCR and image analysis are blobs in Azure Blob Storage and Azure Data Lake Storage (ADLS) Gen2, and image content in Microsoft OneLake. Images can be standalone files or embedded images in a PDF or other files.

ⓘ Note

This skill is bound to Foundry Tools and requires a [billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the [Foundry Tools Standard price](#).

In addition, image extraction is [billable by Azure AI Search](#).

Skill parameters

Parameters are case sensitive.

[] [Expand table](#)

Parameter name	Description
<code>detectOrientation</code>	Detects image orientation. Valid values are <code>true</code> or <code>false</code> .

Parameter name	Description
	This parameter only applies if the legacy OCR version 3.2 API is used.
<code>defaultLanguageCode</code>	Language code of the input text. Supported languages include all of the generally available languages of Azure Vision. You can also specify <code>unk</code> (Unknown). If the language code is unspecified or null, the language is set to English. If the language is explicitly set to <code>unk</code> , all languages found are auto-detected and returned.
<code>lineEnding</code>	The value to use as a line separator. Possible values: "Space", "CarriageReturn", "LineFeed". The default is "Space".

In previous versions, there was a parameter called "textExtractionAlgorithm" to specify extraction of "printed" or "handwritten" text. This parameter is deprecated because the current Read API algorithm extracts both types of text at once. If your skill includes this parameter, you don't need to remove it, but it won't be used during skill execution.

Skill inputs

 [Expand table](#)

Input	Description
<code>name</code>	
<code>image</code>	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure blob indexer when <code>imageAction</code> is set to a value other than <code>none</code> .

Skill outputs

 [Expand table](#)

Output	Description
<code>name</code>	
<code>text</code>	Plain text extracted from the image.
<code>layoutText</code>	Complex type that describes the extracted text and the location where the text was found.

If you call OCR on images embedded in PDFs or other application files, the OCR output will be located at the bottom of the page, after any text that was extracted and processed.

Sample definition

JSON

```
{  
  "skills": [  
    {  
      "description": "Extracts text (plain and structured) from image.",  
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",  
      "context": "/document/normalized_images/*",  
      "defaultLanguageCode": null,  
      "detectOrientation": true,  
      "inputs": [  
        {  
          "name": "image",  
          "source": "/document/normalized_images/*"  
        }  
      ],  
      "outputs": [  
        {  
          "name": "text",  
          "targetName": "myText"  
        },  
        {  
          "name": "layoutText",  
          "targetName": "myLayoutText"  
        }  
      ]  
    }  
  ]  
}
```

Sample text and layoutText output

JSON

```
{  
  "text": "Hello World. -John",  
  "layoutText":  
  {  
    "language" : "en",  
    "text" : "Hello World. -John",  
    "lines" : [  
      {  
        "boundingBox":  
          [ {"x":10, "y":10}, {"x":50, "y":10}, {"x":50, "y":30},{ "x":10, "y":30}],  
        "text": "Hello World."  
      },  
      {  
        "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150, "y":30},  
        {"x":110, "y":30}],  
        "text": "Hello World."  
      }  
    ]  
  }  
}
```

```

        "text": "-John"
    }
],
"words": [
{
    "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150, "y":30},
{"x":110, "y":30}],
    "text": "Hello"
},
{
    "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150, "y":30},
{"x":110, "y":30}],
    "text": "World."
},
{
    "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150, "y":30},
{"x":110, "y":30}],
    "text": "-John"
}
]
}
}

```

Sample: Merging text extracted from embedded images with the content of the document

Document cracking, the first step in skillset execution, separates text and image content. A common use case for Text Merger is merging the textual representation of images (text from an OCR skill, or the caption of an image) into the content field of a document. This is for scenarios where the source document is a PDF or Word document that combines text with embedded images.

The following example skillset creates a *merged_text* field. This field contains the textual content of your document and the OCRed text from each of the images embedded in that document.

Request Body Syntax

JSON
<pre>{ "description": "Extract text from images and merge with content text to produce merged_text", "skills": [{ "description": "Extract text (plain and structured) from image." }] }</pre>

```

"@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
"context": "/document/normalized_images/*",
"defaultLanguageCode": "en",
"detectOrientation": true,
"inputs": [
  {
    "name": "image",
    "source": "/document/normalized_images/*"
  }
],
"outputs": [
  {
    "name": "text"
  }
]
},
{
  "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
  "description": "Create merged_text, which includes all the textual representation of each image inserted at the right location in the content field.",
  "context": "/document",
  "insertPreTag": " ",
  "insertPostTag": " ",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    },
    {
      "name": "itemsToInsert",
      "source": "/document/normalized_images/*/text"
    },
    {
      "name": "offsets",
      "source": "/document/normalized_images/*/contentOffset"
    }
  ],
  "outputs": [
    {
      "name": "mergedText",
      "targetName": "merged_text"
    }
  ]
}
]
}

```

The above skillset example assumes that a normalized-images field exists. To generate this field, set the *imageAction* configuration in your indexer definition to *generateNormalizedImages* as shown below:

JSON

```
{  
    //...rest of your indexer definition goes here ...  
    "parameters": {  
        "configuration": {  
            "dataToExtract": "contentAndMetadata",  
            "imageAction": "generateNormalizedImages"  
        }  
    }  
}
```

See also

- [What is optical character recognition](#)
- [Built-in skills](#)
- [TextMerger skill](#)
- [How to define a skillset](#)
- [Extract text and information from images](#)
- [Create Indexer \(REST\)](#)

Last updated on 11/18/2025

Personally Identifiable Information (PII) Detection cognitive skill

Article • 08/28/2024

The **PII Detection** skill extracts personal information from an input text and gives you the option of masking it. This skill uses the [detection models](#) provided in [Azure AI Language](#).

ⓘ Note

This skill is bound to Azure AI services and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Azure AI services pay-as-you go price](#).

@odata.type

Microsoft.Skills.Text.PIIDetectionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). You can use [Text Split skill](#) for data chunking. Set the page length to 5000 for the best results.

Skill parameters

Parameters are case-sensitive and all are optional.

ⓘ [Expand table](#)

Parameter name	Description
<code>defaultLanguageCode</code>	(Optional) The language code to apply to documents that don't specify language explicitly. If the default language code isn't specified, English (en) is the default language code. See the full list of supported languages .
<code>minimumPrecision</code>	A value between 0.0 and 1.0. If the confidence score (in the <code>piiEntities</code> output) is lower than the set <code>minimumPrecision</code> value, the entity isn't

Parameter name	Description
	returned or masked. The default is 0.0.
maskingMode	A parameter that provides various ways to mask the personal information detected in the input text. The following options are supported: <ul style="list-style-type: none"> "none" (default): No masking occurs and the <code>maskedText</code> output isn't returned. "replace": Replaces the detected entities with the character given in the <code>maskingCharacter</code> parameter. The character is repeated to the length of the detected entity so that the offsets will correctly correspond to both the input text and the output <code>maskedText</code>.
maskingCharacter	The character used to mask the text if the <code>maskingMode</code> parameter is set to <code>replace</code> . The following option is supported: * (default). This parameter can only be <code>null</code> if <code>maskingMode</code> isn't set to <code>replace</code> .
domain	(Optional) A string value, if specified, sets the domain to a subset of the entity categories. Possible values include: "phi" (detect confidential health information only), "none".
piiCategories	(Optional) If you want to specify which entities are detected and returned, use this optional parameter (defined as a list of strings) with the appropriate entity categories. This parameter can also let you detect entities that aren't enabled by default for your document language. See Supported Personally Identifiable Information entity categories for the full list.
modelVersion	(Optional) Specifies the version of the model to use when calling personally identifiable information detection. It defaults to the most recent version when not specified. We recommend you don't specify this value unless it's necessary.

Skill inputs

[+] [Expand table](#)

Input name	Description
languageCode	A string indicating the language of the records. If this parameter isn't specified, the default language code is used to analyze the records. See the full list of supported languages .
text	The text to analyze.

Skill outputs

Output	Description
name	
piiEntities	An array of complex types that contains the following fields: <ul style="list-style-type: none"> • "text" (The actual personally identifiable information as extracted) • "type" • "subType" • "score" (Higher value means it's more likely to be a real entity) • "offset" (into the input text) • "length" <p>See Supported Personally Identifiable Information entity categories for the full list.</p>
maskedText	This output varies depending <code>maskingMode</code> . If <code>maskingMode</code> is <code>replace</code> , output is the string result of the masking performed over the input text, as described by the <code>maskingMode</code> . If <code>maskingMode</code> is <code>none</code> , there's no output.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.PIIDetectionSkill",
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.5,
  "maskingMode": "replace",
  "maskingCharacter": "*",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "piiEntities"
    },
    {
      "name": "maskedText"
    }
  ]
}
```

Sample input

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "text": "Microsoft employee with ssn 859-98-0987 is using our  
awesome API's."  
        }  
    }  
  ]  
}
```

Sample output

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data" :  
        {  
          "piiEntities": [  
            {  
              "text": "859-98-0987",  
              "type": "U.S. Social Security Number (SSN)",  
              "subtype": "",  
              "offset": 28,  
              "length": 11,  
              "score": 0.65  
            }  
          ],  
          "maskedText": "Microsoft employee with ssn ***** is using our  
awesome API's."  
        }  
    }  
  ]  
}
```

The offsets returned for entities in the output of this skill are directly returned from the [Language Service APIs](#), which means if you're using them to index into the original string, you should use the [StringInfo](#) class in .NET in order to extract the correct content. For more information, see [Multilingual and emoji support in Language service features](#).

Errors and warnings

If the language code for the document is unsupported, a warning is returned and no entities are extracted. If your text is empty, a warning is returned. If your text is larger than 50,000 characters, only the first 50,000 characters are analyzed and a warning is issued.

If the skill returns a warning, the output `maskedText` may be empty, which can impact any downstream skills that expect the output. For this reason, be sure to investigate all warnings related to missing output when writing your skillset definition.

See also

- [Built-in skills](#)
 - [How to define a skillset](#)
-

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Sentiment cognitive skill (v3)

The **Sentiment** skill (v3) evaluates unstructured text and for each record, provides sentiment labels (such as "negative", "neutral" and "positive") based on the highest confidence score found by the service at a sentence and document-level. This skill uses the machine learning models provided by version 3 of [Language Service](#) in Foundry Tools. It also exposes [opinion mining capabilities](#), which provides more granular information about the opinions related to attributes of products or services in text.

! Note

This skill is bound to Foundry Tools and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

@odata.type

Microsoft.Skills.Text.V3.SentimentSkill

Data limits

The maximum size of a record should be 5000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the sentiment skill, use the [Text Split skill](#).

Skill parameters

Parameters are case sensitive.

[] Expand table

Parameter Name	Description
defaultLanguageCode	(optional) The language code to apply to documents that don't specify language explicitly. See the full list of supported languages .
modelVersion	(optional) Specifies the version of the model to use when calling sentiment analysis. It will default to the most recent version when not specified. We recommend you do not specify this value unless it's necessary.
includeOpinionMining	If set to <code>true</code> , enables the opinion mining feature , which allows aspect-based

Parameter Name	Description
	sentiment analysis to be included in your output results. Defaults to <code>false</code> .

Skill inputs

[+] Expand table

Input Name	Description
<code>text</code>	The text to be analyzed.
<code>languageCode</code>	(optional) A string indicating the language of the records. If this parameter is not specified, the default value is "en". See the full list of supported languages .

Skill outputs

[+] Expand table

Output Name	Description
<code>sentiment</code>	A string value that represents the sentiment label of the entire analyzed text (either positive, neutral or negative).
<code>confidenceScores</code>	A complex type with three double values, one for the positive rating, one for the neutral rating, and one for the negative rating. Values range from 0 to 1.00, where 1.00 represents the highest possible confidence in a given label assignment.
<code>sentences</code>	A collection of complex types that breaks down the sentiment of the text sentence by sentence. This is also where opinion mining results are returned in the form of targets and assessments if <code>includeOpinionMining</code> is set to <code>true</code> .

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.SentimentSkill",
  "context": "/document",
  "includeOpinionMining": true,
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ]
}
```

```
        },
        {
            "name": "languageCode",
            "source": "/document/languageCode"
        }
    ],
    "outputs": [
        {
            "name": "sentiment",
            "targetName": "sentiment"
        },
        {
            "name": "confidenceScores",
            "targetName": "confidenceScores"
        },
        {
            "name": "sentences",
            "targetName": "sentences"
        }
    ]
}
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "I had a terrible time at the hotel. The staff was rude and
the food was awful.",
                "languageCode": "en"
            }
        }
    ]
}
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "sentiment": "negative",

```

```
"confidenceScores": {
    "positive": 0.0,
    "neutral": 0.0,
    "negative": 1.0
},
"sentences": [
    {
        "text": "I had a terrible time at the hotel.",
        "sentiment": "negative",
        "confidenceScores": {
            "positive": 0.0,
            "neutral": 0.0,
            "negative": 1.0
        },
        "offset": 0,
        "length": 35,
        "targets": [],
        "assessments": []
    },
    {
        "text": "The staff was rude and the food was awful.",
        "sentiment": "negative",
        "confidenceScores": {
            "positive": 0.0,
            "neutral": 0.0,
            "negative": 1.0
        },
        "offset": 36,
        "length": 42,
        "targets": [
            {
                "text": "staff",
                "sentiment": "negative",
                "confidenceScores": {
                    "positive": 0.0,
                    "neutral": 0.0,
                    "negative": 1.0
                },
                "offset": 40,
                "length": 5,
                "relations": [
                    {
                        "relationType": "assessment",
                        "ref": "#/documents/0/sentences/1/assessments/0"
                    }
                ]
            },
            {
                "text": "food",
                "sentiment": "negative",
                "confidenceScores": {
                    "positive": 0.0,
                    "neutral": 0.0,
                    "negative": 1.0
                }
            }
        ]
    }
]
```

```

        },
        "offset": 63,
        "length": 4,
        "relations": [
            {
                "relationType": "assessment",
                "ref": "#/documents/0/sentences/1/assessments/1",
                "text": "rude"
            }
        ],
        "assessments": [
            {
                "text": "rude",
                "sentiment": "negative",
                "confidenceScores": {
                    "positive": 0.0,
                    "neutral": 0.0,
                    "negative": 1.0
                },
                "offset": 50,
                "length": 4,
                "isNegated": false
            },
            {
                "text": "awful",
                "sentiment": "negative",
                "confidenceScores": {
                    "positive": 0.0,
                    "neutral": 0.0,
                    "negative": 1.0
                },
                "offset": 72,
                "length": 5,
                "isNegated": false
            }
        ]
    }
}

```

Warning cases

If your text is empty, a warning is generated and no sentiment results are returned. If a language is not supported, a warning is generated and no sentiment results are returned.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Last updated on 11/18/2025

Text Translation cognitive skill

The **Text Translation** skill evaluates text and, for each record, returns the text translated to the specified target language. This skill uses the [Translator Text API v3.0](#) available in Foundry Tools.

This capability is useful if you expect that your documents may not all be in one language, in which case you can normalize the text to a single language before indexing for search by translating it. It's also useful for localization use cases, where you might want to have copies of the same text available in multiple languages.

The [Translator Text API v3.0](#) is a non-regional Foundry Tool, meaning that your data isn't guaranteed to stay in the same region as your Azure AI Search or attached Microsoft Foundry resource.

(!) Note

This skill is bound to Foundry Tools and requires [a billable resource](#) for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

When using this skill, all documents in the source are processed and billed for translation, even if the source and target languages are the same. This behavior is useful for multi-language support within the same document, but it can result in unnecessary processing. To avoid unexpected billing charges from documents that don't need processing, move them out of the data source container prior to running the skill.

@odata.type

Microsoft.Skills.Text.TranslationSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the text translation skill, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case sensitive.

[Expand table](#)

Inputs	Description
defaultToLanguageCode	(Required) The language code to translate documents into for documents that don't specify the "to" language explicitly. See the full list of supported languages .
defaultFromLanguageCode	(Optional) The language code to translate documents from for documents that don't specify the "from" language explicitly. If the defaultFromLanguageCode isn't specified, the automatic language detection provided by the Translator Text API will be used to determine the "from" language. See the full list of supported languages .
suggestedFrom	(Optional) The language code to translate documents from if <code>fromLanguageCode</code> or <code>defaultFromLanguageCode</code> are unspecified, and the automatic language detection is unsuccessful. If the suggestedFrom language isn't specified, English (en) will be used as the suggestedFrom language. See the full list of supported languages .

Skill inputs

[Expand table](#)

Input name	Description
text	The text to be translated.
toLanguageCode	A string indicating the language the text should be translated to. If this input isn't specified, the defaultToLanguageCode will be used to translate the text. See the full list of supported languages .
fromLanguageCode	A string indicating the current language of the text. If this parameter isn't specified, the defaultFromLanguageCode (or automatic language detection if the defaultFromLanguageCode isn't provided) will be used to translate the text. See the full list of supported languages .

Skill outputs

[Expand table](#)

Output name	Description
translatedText	The string result of the text translation from the translatedFromLanguageCode to the translatedToLanguageCode.
translatedToLanguageCode	A string indicating the language code the text was translated to. Useful if you're translating to multiple languages and want to be able to keep track of which text is which language.
translatedFromLanguageCode	A string indicating the language code the text was translated from. Useful if you opted for the automatic language detection option as this output will give you the result of that detection.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",
  "defaultToLanguageCode": "fr",
  "suggestedFrom": "en",
  "context": "/document",
  "inputs": [
    {
      "name": "text",
      "source": "/document/text"
    }
  ],
  "outputs": [
    {
      "name": "translatedText",
      "targetName": "translatedText"
    },
    {
      "name": "translatedFromLanguageCode",
      "targetName": "translatedFromLanguageCode"
    },
    {
      "name": "translatedToLanguageCode",
      "targetName": "translatedToLanguageCode"
    }
  ]
}
```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "We hold these truths to be self-evident, that all men are created equal."
      }
    },
    {
      "recordId": "2",
      "data": {
        "text": "Estamos muy felices de estar con ustedes."
      }
    }
  ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "translatedText": "Nous tenons ces vérités pour évidentes, que tous les hommes sont créés égaux.",
        "translatedFromLanguageCode": "en",
        "translatedToLanguageCode": "fr"
      }
    },
    {
      "recordId": "2",
      "data": {
        "translatedText": "Nous sommes très heureux d'être avec vous.",
        "translatedFromLanguageCode": "es",
        "translatedToLanguageCode": "fr"
      }
    }
  ]
}
```

Errors and warnings

If you provide an unsupported language code for either the "to" or "from" language, an error is generated, and text isn't translated. If your text is empty, a warning will be produced. If your text is larger than 50,000 characters, only the first 50,000 characters will be translated, and a warning will be issued.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Last updated on 11/18/2025

Azure Content Understanding skill

The Azure Content Understanding skill uses [document analyzers](#) from [Azure Content Understanding in Foundry Tools](#) to analyze unstructured documents and other content types, generating organized, searchable outputs that can be integrated into automation workloads. This skill extracts both text and images, including location metadata that preserves each image's position within the document. Image proximity to related content is especially useful for [multimodal search](#), [agentic retrieval](#), and [retrieval-augmented generation](#) (RAG).

You can use the Azure Content Understanding skill for both content extraction and chunking. There's no need to use the Text Split skill in your skillset. This skill implements the same interface as the Document Layout skill, which uses the [Azure Document Intelligence in Foundry Tools layout model](#) when `outputFormat` is set to `text`. However, the Azure Content Understanding skill offers several advantages over the Document Layout skill:

- Tables and figures are output in Markdown format, making them easier for large language models (LLMs) to understand. In contrast, the Document Layout skill outputs tables and figures as plain text, which can result in information loss.
- For tables that span multiple pages, the Document Layout skill extracts tables page by page. The Azure Content Understanding skill can recognize and extract cross-page tables as a single unit.
- The Document Layout skill restricts chunks to a single page, but semantic units, such as cross-page tables, shouldn't be limited by page boundaries. The Azure Content Understanding skill allows chunks to span multiple pages.
- The Azure Content Understanding skill is more cost effective than the Document Layout skill because the Content Understanding API is less expensive.

The Azure Content Understanding skill is bound to a [billable Microsoft Foundry resource](#). Unlike other Azure AI resource skills, such as the [Document Layout skill](#), the Azure Content Understanding skill doesn't provide 20 free documents per indexer per day. Execution of this skill is charged at the [Azure Content Understanding price](#).

💡 Tip

You can use the Azure Content Understanding skill in a skillset that also performs image verbalization and chunk vectorization. In the following tutorials, replace the Document Layout skill with the Azure Content Understanding skill.

- [Tutorial: Verbalize images from a structured document layout](#)

- [Tutorial: Vectorize from a structured document layout](#)

Limitations

The Azure Content Understanding skill has the following limitations:

- This skill isn't suitable for large documents requiring more than five minutes of processing in the Content Understanding document analyzer. The skill times out, but charges still apply to the Foundry resource that's attached to the skillset. Ensure documents are optimized to stay within processing limits to avoid unnecessary costs.
- This skill calls the Azure Content Understanding document analyzer, so all documented [service behaviors for different document types](#) apply to its output. For example, Word (DOCX) and PDF files might produce different results due to differences in how images are handled. If consistent image behavior across DOCX and PDF is required, consider converting documents to PDF or reviewing the [multimodal search documentation](#) for alternative approaches.

Supported regions

The Azure Content Understanding skill calls the [Content Understanding 2025-05-01-preview REST API](#). Your Foundry resource must be in a supported region, which is described in [Azure Content Understanding region and language support](#).

Your search service can be in any [supported Azure AI Search region](#). When your Foundry resource and Azure AI Search service aren't in the same region, cross-region network latency impacts your indexer's performance.

Supported file formats

The Azure Content Understanding skill recognizes the following file formats:

- .PDF
- .JPEG
- .JPG
- .PNG
- .BMP
- .HEIF
- .TIFF
- .DOCX

- .XLSX
- .PPTX
- .HTML
- .TXT
- .MD
- .RTF
- .EML

Supported languages

For printed text, see [Azure Content Understanding region and language support](#).

@odata.type

Microsoft.Skills.Util.ContentUnderstandingSkill

Data limits

- Even when the file size for analyzing documents is within the 200 MB limit, as described in the [Azure Content Understanding service quotas and limits](#), indexing is still subject to the [indexer limits](#) of your search service tier.
- Image dimensions must be between 50 pixels x 50 pixels or 10,000 pixels x 10,000 pixels.
- If your PDFs are password locked, remove the lock before you run the indexer.

Skill parameters

Parameters are case sensitive.

[] Expand table

Parameter name	Allowed values	Description
extractionOptions	["images"], ["images", "locationMetadata"], ["locationMetadata"]	Identify any extra content extracted from the document. Define an array of enums that correspond to the content to be included in the output. For example, if <code>extractionOptions</code> is ["images", "locationMetadata"], the output includes images and location metadata that provides page location and visual information related to where the content was extracted.

Parameter name	Allowed values	Description
chunkingProperties	See the following table.	Options that encapsulate how to chunk text content.

[\[+\] Expand table](#)

chunkingProperties	Allowed values	Description
parameters		
unit	Characters is the only allowed value. The chunk length is measured in characters rather than words or tokens.	Controls the cardinality of the chunk unit.
maxLength	An integer between 300 and 50000.	The maximum chunk length in characters as measured by <code>String.Length</code> .
overlapLength	Integer. The value must be less than half of the <code>maxLength</code> .	The length of overlap provided between two text chunks.

Skill inputs

[\[+\] Expand table](#)

Input name	Description
file_data	The file from which content should be extracted.

The `file_data` input must be an object defined as:

JSON

```
{
  "$type": "file",
  "data": "BASE64 encoded string of the file"
}
```

Alternatively, it can be defined as:

JSON

```
{
  "$type": "file",
  "url": "URL to download the file",
  "sasToken": "OPTIONAL: SAS token for authentication if the provided URL is for a
```

```
file in blob storage"  
}
```

The file reference object can be generated in one of following ways:

- Setting the `allowSkillsetToReadFileData` parameter on your indexer definition to `true`. This setting creates a `/document/file_data` path that's an object representing the original file data downloaded from your blob data source. This parameter only applies to files in Azure Blob Storage.
- Having a custom skill returning a JSON object definition that provides `$type`, `data`, or `url` and `sastoken`. The `$type` parameter must be set to `file`, and `data` must be the base 64-encoded byte array of the file content. The `url` parameter must be a valid URL with access to download the file at that location.

Skill outputs

 Expand table

Output name	Description
<code>text_sections</code>	A collection of text chunk objects. Each chunk can span multiple pages (factoring in any more chunking configured). The text chunk object includes <code>locationMetadata</code> if applicable.
<code>normalized_images</code>	Only applies if <code>extractionOptions</code> includes <code>images</code> . A collection of images that were extracted from the document, including <code>locationMetadata</code> if applicable.

Example

This example demonstrates how to output text content in fixed-sized chunks and extract images along with location metadata from the document.

Sample definition that includes image and metadata extraction

JSON

```
{  
  "skills": [  
    {  
      "description": "Analyze a document",  
      "@odata.type": "#Microsoft.Skills.Util.ContentUnderstandingSkill",  
      "inputs": [  
        {  
          "name": "text",  
          "type": "Text",  
          "value": "The quick brown fox jumps over the lazy dog."  
        }  
      ],  
      "outputs": [  
        {  
          "name": "text_sections",  
          "type": "Text",  
          "value": "The quick brown fox jumps over the lazy dog."  
        },  
        {  
          "name": "normalized_images",  
          "type": "Image",  
          "value": "The quick brown fox jumps over the lazy dog."  
        }  
      ]  
    }  
  ]  
}
```

```
"context": "/document",
"extractionOptions": ["images", "locationMetadata"],
"chunkingProperties": {
    "unit": "characters",
    "maxLength": 1325,
    "overlapLength": 0
},
"inputs": [
    {
        "name": "file_data",
        "source": "/document/file_data"
    }
],
"outputs": [
    {
        "name": "text_sections",
        "targetName": "text_sections"
    },
    {
        "name": "normalized_images",
        "targetName": "normalized_images"
    }
]
}
```

Sample output

JSON

```
{  
  "text_sections": [  
    {  
      "id": "1_d4545398-8df1-409f-acbb-f605d851ae85",  
      "content": "What is Azure Content Understanding (preview)?  
09/16/2025Important. Azure AI Content Understanding is available in preview. Public  
preview releases provide early access to features that are in active development..  
Features, approaches, and processes can change or have limited capabilities, before  
General Availability (GA).. For more information, see Supplemental Terms of Use for  
Microsoft Azure PreviewsAzure Content Understanding is a Foundry Tool that uses  
generative AI to process/ingest content of many types (documents, images, videos,  
and audio) into a user-defined output format.Content Understanding offers a  
streamlined process to reason over large amounts of unstructured data, accelerating  
time-to-value by generating an output that can be integrated into automation and  
analytical workflows.  
<figure>\n\nInputs\n\nAnalyzers\n\nOutput\n\n0\nSearch\n\nContent  
Extraction\n\nField  
Extraction\n\nDocuments\n\nNew\n\nAgents\n\nPreprocessing\n\nEnrichments\n\nReasonin  
g\n\nImage\n\nNormalization\n(resolution,\nformats)\n\nSpeaker\n\nrecognition\n\nGen  
AI\nContext\nwindows\n\nPostprocessing\nConfidence\nscores\nGrounding\nNormalization  
\nMulti-file input\nReference data\n\nDatabases\n\nVideo\n\nOrientation /\nde-  
skew\n\nLayout and\nstructure\n\nPrompt
```

tuning\n\nStructured\noutput\n\nAudio\n\nFace grouping\n\nMarkdown or JSON schema\n\nCopilots\n\nApps\n\n\\+\n\nFaurIC\n\n</figure>,

 "locationMetadata": {

 "pageNumberFrom": 1,

 "pageNumberTo": 1,

 "ordinalPosition": 0,

 "source":

 "D(1,0.6348,0.3598,7.2258,0.3805,7.223,1.2662,0.632,1.2455);D(1,0.6334,1.3758,1.3896,1.3738,1.39,1.5401,0.6338,1.542);D(1,0.8104,2.0716,1.8137,2.0692,1.8142,2.2669,0.8109,2.2693);D(1,1.0228,2.5023,7.6222,2.5029,7.6221,3.0075,1.0228,3.0069);D(1,1.0216,3.1121,7.3414,3.1057,7.342,3.6101,1.0221,3.6165);D(1,1.0219,3.7145,7.436,3.7048,7.4362,3.9006,1.0222,3.9103);D(1,0.6303,4.3295,7.7875,4.3236,7.7879,4.812,0.6307,4.8179);D(1,0.6304,5.0295,7.8065,5.0303,7.8064,5.7858,0.6303,5.7849);D(1,0.635,5.9572,7.8544,5.9573,7.8562,8.6971,0.6363,8.6968);D(1,0.6381,9.1451,5.2731,9.1476,5.2729,9.4829,0.6379,9.4803)"

 }

},

...

{

 "id": "2_e0e57fd4-e835-4879-8532-73a415e47b0b",

 "content": "

<table>\n<tr>\n<th>Application</th>\n<th>Description</th>\n</tr>\n<tr>\n<td>Post-call analytics</td>\n<td>Businesses and call centers can generate insights from call recordings to track key KPIs, improve product experience, generate business insights, create differentiated customer experiences, and answer queries faster and more accurately.

</td>\n</tr>\n<tr>\n<th>Application</th>\n<th>Description</th>\n</tr>\n<tr>\n<td>Media asset management</td>\n<td>Software and media vendors can use Content Understanding to extract richer, targeted information from videos for media asset management solutions.</td>\n</tr>\n<tr>\n<td>Tax automation</td>\n<td>Tax preparation companies can use Content Understanding to generate a unified view of information from various documents and create comprehensive tax returns.

</td>\n</tr>\n<tr>\n<td>Chart understanding</td>\n<td>Businesses can enhance chart understanding by automating the analysis and interpretation of various types of charts and diagrams using Content Understanding.</td>\n</tr>\n<tr>\n<td>Mortgage application processing</td>\n<td>Analyze supplementary supporting documentation and mortgage applications to determine whether a prospective home buyer provided all the necessary documentation to secure a mortgage.</td>\n</tr>\n<tr>\n<td>Invoice contract verification</td>\n<td>Review invoices and contr",

 "locationMetadata": {

 "pageNumberFrom": 2,

 "pageNumberTo": 3,

 "ordinalPosition": 3,

 "source":

 "D(2,0.6438,9.2645,7.8576,9.2649,7.8565,10.5199,0.6434,10.5194);D(3,0.6494,0.3919,7.8649,0.3929,7.8639,4.3254,0.6485,4.3232)"

 }

...

}

],

 "normalized_images": [

 {

 "id": "1_335140f1-9d31-4507-8916-2cde758639cb",

 "data": "aW1hZ2UgMSBkYXRh",

 "imagePath":

```

"aHR0cHM6Ly9henNyb2xsaw5nLmJsb2IuY29yZS53aW5kb3dzLm5ldC9tdWx0aW1vZGFsaXR5L0NVLnBkZg2
/normalized_images_0.jpg",
    "locationMetadata": {
        "pageNumberFrom": 1,
        "pageNumberTo": 1,
        "ordinalPosition": 0,
        "source": "D(1,0.635,5.9572,7.8544,5.9573,7.8562,8.6971,0.6363,8.6968)"
    }
},
{
    "id": "3_699d33ac-1a1b-4015-9cbd-eb8bfff2e6b4",
    "data": "aW1hZ2UgMiBkYXRh",
    "imagePath": "aHR0cHM6Ly9henNyb2xsaw5nLmJsb2IuY29yZS53aW5kb3dzLm5ldC9tdWx0aW1vZGFsaXR5L0NVLnBkZg2
/normalized_images_1.jpg",
    "locationMetadata": {
        "pageNumberFrom": 3,
        "pageNumberTo": 3,
        "ordinalPosition": 1,
        "source": "D(3,0.6353,5.2142,7.8428,5.218,7.8443,8.4631,0.6363,8.4594)"
    }
}
]
}

```

`locationMetadata` is based on the `source` property provided by Azure Content Understanding. For information about how visual position of the element in the file is encoded, see [Document analysis: Extract structured content](#).

`imagePath` represents the relative path of a stored image. If the knowledge store file projection is configured in the skillset, this path matches the relative path of the image stored in the knowledge store.

Related content

- [What is Azure Content Understanding \(preview\)?](#)
- [Built-in skills](#)
- [Create a skillset](#)
- [Indexers - Create \(REST API\)](#)

Azure OpenAI Embedding skill

The Azure OpenAI Embedding skill connects to an embedding model deployed to your [Azure OpenAI in Foundry Models](#) resource or [Microsoft Foundry](#) project to generate embeddings during indexing. Your data is processed in the [Geo ↗](#) where your model is deployed.

The [Import data \(new\) wizard](#) in the Azure portal uses the Azure OpenAI Embedding skill to vectorize content. You can run the wizard and review the generated skillset to see how the wizard builds the skill for embedding models.

!Note

This skill is bound to Azure OpenAI and is charged at the [Azure OpenAI Standard price ↗](#).

Prerequisites

- An [Azure OpenAI in Foundry Models resource](#) or [Foundry project](#).
 - Your Azure OpenAI resource must have a [custom subdomain](#), such as `https://<resource-name>.openai.azure.com`. You can find this endpoint on the **Keys and Endpoint** page in the Azure portal and use it for the `resourceUri` property in this skill.
 - The [parent resource](#) of your Foundry project provides access to multiple endpoints, including `https://<resource-name>.openai.azure.com`, `https://<resource-name>.services.ai.azure.com`, and `https://<resource-name>.cognitiveservices.azure.com`. You can find these endpoints on the **Keys and Endpoint** page in the Azure portal and use any of them for the `resourceUri` property in this skill.
- An Azure OpenAI embedding model deployed to your resource or project. For supported models, see the [Skill parameters](#) section.

@odata.type

`Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill`

Data limits

The maximum size of a text input should be 8,000 tokens. If input exceeds the maximum allowed, the model throws an invalid request error. For more information, see the [tokens](#) key concept in the Azure OpenAI documentation. Consider using the [Text Split skill](#) if you need data chunking.

Skill parameters

Parameters are case sensitive.

[] [Expand table](#)

Inputs	Description
<code>resourceUri</code>	(Required) The URI of the model provider. Supported domains are: <ul style="list-style-type: none">• <code>openai.azure.com</code>• <code>services.ai.azure.com</code>• <code>cognitiveservices.azure.com</code> This field is required if your resource is deployed behind a private endpoint or uses virtual network (VNet) integration. Azure API Management endpoints are supported with URL <code>https://<resource-name>.azure-api.net</code> . Shared private links aren't supported for API Management endpoints.
<code>apiKey</code>	The secret key used to access the model. If you provide a key, leave <code>authIdentity</code> empty. If you set both <code>apiKey</code> and <code>authIdentity</code> , the <code>apiKey</code> is used on the connection.
<code>deploymentId</code>	(Required) The ID of the deployed Azure OpenAI embedding model. This is the deployment name you specified when you deployed the model.
<code>authIdentity</code>	A user-managed identity used by the search service for the connection. You can use either a system- or user-managed identity . To use a system-managed identity, leave <code>apiKey</code> and <code>authIdentity</code> blank. The system-managed identity is used automatically. A managed identity must have Cognitive Services OpenAI User permissions to send text to Azure OpenAI.
<code>modelName</code>	(Required) The name of the Azure OpenAI model deployed at the specified <code>deploymentId</code> . Supported values are: <ul style="list-style-type: none">• <code>text-embedding-ada-002</code>• <code>text-embedding-3-large</code>• <code>text-embedding-3-small</code>
<code>dimensions</code>	(Optional) The dimensions of embeddings that you want to generate, assuming the model supports a range of dimensions . The default is the maximum dimensions for each model. For skillsets created with REST API versions prior to the 2023-10-01-preview, the

Inputs	Description
	dimensions are fixed at 1536. If you set the <code>dimensions</code> property in this skill, set the <code>dimensions</code> property on the vector field definition to the same value.

Supported dimensions by `modelName`

The supported dimensions for an Azure OpenAI Embedding skill depend on the `modelName` that is configured.

[] [Expand table](#)

<code>modelName</code>	Minimum dimensions	Maximum dimensions
text-embedding-ada-002	1536	1536
text-embedding-3-large	1	3072
text-embedding-3-small	1	1536

Skill inputs

[] [Expand table](#)

Input	Description
<code>text</code>	The input text to be vectorized. If you're using data chunking, the source might be <code>/document/pages/*</code> .

Skill outputs

[] [Expand table](#)

Output	Description
<code>embedding</code>	Vectorized embedding for the input text.

Sample definition

Consider a record that has the following fields:

JSON

```
{  
    "content": "Microsoft released Windows 10."  
}
```

Then your skill definition might look like this:

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",  
    "description": "Connects a deployed embedding model.",  
    "resourceUri": "https://my-demo-openai-eastus.openai.azure.com/",  
    "deploymentId": "my-text-embedding-ada-002-model",  
    "modelName": "text-embedding-ada-002",  
    "dimensions": 1536,  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "embedding"  
        }  
    ]  
}
```

Sample output

For the given input text, a vectorized embedding output is produced.

JSON

```
{  
    "embedding": [  
        0.018990106880664825,  
        -0.0073809814639389515,  
        ....  
        0.021276434883475304,  
    ]  
}
```

The output resides in memory. To send this output to a field in the search index, you must define an [outputFieldMapping](#) that maps the vectorized embedding output (which is an array) to a [vector field](#). Assuming the skill output resides in the document's **embedding** node, and

`content_vector` is the field in the search index, the `outputFieldMapping` in indexer should look like:

JSON

```
"outputFieldMappings": [
  {
    "sourceFieldName": "/document/embedding/*",
    "targetFieldName": "content_vector"
  }
]
```

Best practices

The following are some best practices you need to consider when utilizing this skill:

- If you are hitting your Azure OpenAI TPM (Tokens per minute) limit, consider the [quota limits advisory](#) so you can address accordingly. Refer to the [Azure OpenAI monitoring](#) documentation for more information about your Azure OpenAI instance performance.
- The Azure OpenAI embeddings model deployment you use for this skill should be ideally separate from the deployment used for other use cases, including the [query vectorizer](#). This helps each deployment to be tailored to its specific use case, leading to optimized performance and identifying traffic from the indexer and the index embedding calls easily.
- Your Azure OpenAI instance should be in the same region or at least geographically close to the region where your AI Search service is hosted. This reduces latency and improves the speed of data transfer between the services.
- If you have a larger than default Azure OpenAI TPM (Tokens per minute) limit as published in [quotas and limits](#) documentation, open a [support case](#) with the Azure AI Search team, so this can be adjusted accordingly. This helps your indexing process not being unnecessarily slowed down by the documented default TPM limit, if you have higher limits.
- For examples and working code samples using this skill, see the following links:
 - [Integrated vectorization \(Python\)](#)
 - [Integrated vectorization \(C#\)](#)
 - [Integrated vectorization \(Java\)](#)

Errors and warnings

Condition	Result
Null or invalid URI	Error
Null or invalid deploymentID	Error
Text is empty	Warning
Text is larger than 8,000 tokens	Error

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [How to define output fields mappings](#)

Last updated on 11/18/2025

GenAI Prompt skill

!**Note**

This feature is currently in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

The **GenAI (Generative AI) Prompt** skill executes a *chat completion* request against a large language model (LLM) deployed in [Azure OpenAI in Foundry Models](#) or [Microsoft Foundry](#). Use this skill to create new information that can be indexed and stored as searchable content.

Here are some examples of how the GenAI prompt skill can help you create content:

- Verbalize images
- Summarize large passages of text
- Simplify complex content
- Perform any other task that you can articulate in a prompt

The GenAI Prompt skill is available in the [latest preview REST API](#). This skill supports text, image, and multimodal content, such as a PDF that contains text and images.

💡 **Tip**

It's common to use this skill combined with a data chunking skill. The following tutorials demonstrate image verbalization with two different data chunking techniques:

- [Tutorial: Verbalize images using generative AI](#)
- [Tutorial: Verbalize images from a structured document layout](#)

Supported models

- You can use any [chat completion inference model](#) deployed in Foundry, such as GPT models, Deepseek R#, Llama-4-Mavericj, and Cohere-command-r. For GPT models specifically, only the chat completions API endpoints are supported. Endpoints using the Azure OpenAI Responses API (containing `/openai/responses` in the URI) aren't currently compatible.
- For image verbalization, the model you use to analyze the image determines what image formats are supported.

- For GPT-5 models, the `temperature` parameter is not supported in the same way as previous models. If defined, it must be set to `1.0`, as other values will result in errors.
- Billing is based on the pricing of the model you use.

(!) Note

The search service connects to your model over a public endpoint, so there are no region location requirements, but if you're using an all-up Azure solution, you should check the [Azure AI Search regions](#) and the [Azure OpenAI model regions](#) to find suitable pairs, especially if you have data residency requirements.

Prerequisites

- An [Azure OpenAI in Foundry Models resource](#) or [Foundry project](#).
- A [supported model](#) deployed to your resource or project.
 - For Azure OpenAI, copy the endpoint with the `openai.azure.com` domain from the **Keys and Endpoint** page in the Azure portal. Use this endpoint for the `Uri` parameter in this skill.
 - For Foundry, copy the target URI for the deployment from the **Models** page in the Foundry portal. Use this endpoint for the `Uri` parameter in this skill.
- Authentication can be key-based with an API key from your Foundry or Azure OpenAI resource. However, we recommend role-based access using a [search service managed identity](#) assigned to a role.
 - On Azure OpenAI, assign [Cognitive Services OpenAI User](#) to the managed identity.
 - On Foundry, assign [Azure AI User](#) to the managed identity.

@odata.type

```
#Microsoft.Skills.Custom.ChatCompletionSkill
```

Data limits

[+] [Expand table](#)

Limit	Notes
<code>maxTokens</code>	Default is 1024 if omitted. Maximum value is model-dependent.
Request time-out	30 seconds (default). Override with the <code>timeout</code> property (<code>PT##S</code>).
Images	Base 64-encoded images and image URLs are supported. Size limit is model-dependent.

Skill parameters

 [Expand table](#)

Property	Type	Required	Notes
<code>uri</code>	string	Yes	Public endpoint of the deployed model. Supported domains are: <ul style="list-style-type: none"> • <code>openai.azure.com</code> • <code>services.ai.azure.com</code> • <code>cognitiveservices.azure.com</code>
<code>apiKey</code>	string	Cond.*	Secret key for the model. Leave blank when using managed identity.
<code>authIdentity</code>	string	Cond.*	User-assigned managed identity client ID (<i>Azure OpenAI only</i>). Leave blank to use the system-assigned identity.
<code>commonModelParameters</code>	object	No	Standard generation controls such as <code>temperature</code> , <code>maxTokens</code> , etc.
<code>extraParameters</code>	object	No	Open dictionary passed through to the underlying model API.
<code>extraParametersBehavior</code>	string	No	<code>"pass-through"</code> <code>"drop"</code> <code>"error"</code> (default <code>"error"</code>).
<code>responseFormat</code>	object	No	Controls whether the model returns <code>text</code> , a free-form <code>JSON object</code> , or a strongly typed <code>JSON schema</code> . <code>responseFormat</code> payload examples: { <code>responseFormat: { type: text }</code> }, { <code>responseFormat: { type: json_object }</code> }, { <code>responseFormat: { type: json_schema }</code> }

* Exactly one of `apiKey`, `authIdentity`, or the service's **system-assigned** identity must be used.

`commonModelParameters` defaults

[Expand table](#)

Parameter	Default
<code>model</code>	(deployment default)
<code>frequencyPenalty</code>	0
<code>presencePenalty</code>	0
<code>maxTokens</code>	1024
<code>temperature</code>	0.7
<code>seed</code>	<code>null</code>
<code>stop</code>	<code>null</code>

Skill inputs

[Expand table](#)

Input name	Type	Required	Description
<code>systemMessage</code>	string	Yes	System-level instruction (ex: "You are a helpful assistant.").
<code>userMessage</code>	string	Yes	User prompt.
<code>text</code>	string	No	Optional text appended to <code>userMessage</code> (text-only scenarios).
<code>image</code>	string (Base 64 data-URL)	No	Adds an image to the prompt (multimodal models only).
<code>imageDetail</code>	string (<code>low</code> <code>high</code> <code>auto</code>)	No	Fidelity hint for Azure OpenAI multimodal models.

Skill outputs

[Expand table](#)

Output name	Type	Description
<code>response</code>	string or JSON object	Model output in the format requested by <code>responseFormat.type</code> .

Output name	Type	Description
usageInformation	JSON object	Token counts and echo of model parameters.

Sample definitions

Text-only summarization

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",
  "name": "Summarizer",
  "description": "Summarizes document content.",
  "context": "/document",
  "timeout": "PT30S",
  "inputs": [
    { "name": "text", "source": "/document/content" },
    { "name": "systemMessage", "source": "'You are a concise AI assistant.'" },
    { "name": "userMessage", "source": "'Summarize the following text:'" }
  ],
  "outputs": [ { "name": "response" } ],
  "uri": "https://demo.openai.azure.com/openai/deployments/gpt-4o/chat/completions",
  "apiKey": "<api-key>",
  "commonModelParameters": { "temperature": 0.3 }
}
```

Text + image description

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",
  "name": "Image Describer",
  "context": "/document/normalized_images/*",
  "inputs": [
    { "name": "image", "source": "/document/normalized_images/*/data" },
    { "name": "imageDetail", "source": "=high" },
    { "name": "systemMessage", "source": "'You are a useful AI assistant.'" },
    { "name": "userMessage", "source": "'Describe this image:'" }
  ],
  "outputs": [ { "name": "response" } ],
  "uri": "https://demo.openai.azure.com/openai/deployments/gpt-4o/chat/completions",
  "authIdentity": "11111111-2222-3333-4444-555555555555",
  "responseFormat": { "type": "text" }
}
```

Structured numerical fact-finder

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.ChatCompletionSkill",  
    "name": "NumericalFactFinder",  
    "context": "/document",  
    "inputs": [  
        { "name": "systemMessage", "source": "'You are an AI assistant that helps  
people find information.'"},  
        { "name": "userMessage", "source": "'Find all the numerical data and put it in  
the specified fact format.'"},  
        { "name": "text", "source": "/document/content" }  
    ],  
    "outputs": [ { "name": "response" } ],  
    "uri": "https://demo.openai.azure.com/openai/deployments/gpt-4o/chat/completions",  
    "apiKey": "<api-key>",  
    "responseFormat": {  
        "type": "json_schema",  
        "jsonSchemaProperties": {  
            "name": "NumericalFactObj",  
            "strict": true,  
            "schema": {  
                "type": "object",  
                "properties": {  
                    "facts": {  
                        "type": "array",  
                        "items": {  
                            "type": "object",  
                            "properties": {  
                                "number": { "type": "number" },  
                                "fact": { "type": "string" }  
                            },  
                            "required": [ "number", "fact" ]  
                        }  
                    }  
                },  
                "required": [ "facts" ],  
                "additionalProperties": false  
            }  
        }  
    }  
}
```

Sample output (truncated)

JSON

```
{  
    "response": {
```

```

    "facts": [
      { "number": 32.0, "fact": "Jordan scored 32 points per game in 1986-87." },
      { "number": 6.0, "fact": "He won 6 NBA championships." }
    ],
  },
  "usageInformation": {
    "usage": {
      "completion_tokens": 203,
      "prompt_tokens": 248,
      "total_tokens": 451
    }
  }
}

```

Best practices

- Chunk long documents with the **Text Split** skill to stay within the model's context window.
- For high-volume indexing, dedicate a separate model deployment to this skill so that token quotas for query-time RAG workloads remain unaffected.
- To minimize latency, co-locate the model and your Azure AI Search service in the same Azure region.
- Use `responseFormat.json_schema` with **GPT-4o** for reliable structured extraction and easier mapping to index fields.
- Monitor token usage and submit **quota-increase requests** if the indexer saturates your Tokens per Minute (TPM) limits.

Errors and warnings

[] Expand table

Condition	Result
Missing or invalid <code>uri</code>	Error
No authentication method specified	Error
Both <code>apiKey</code> and <code>authIdentity</code> supplied	Error
Unsupported model for multimodal prompt	Error
Input exceeds model token limit	Error
Model returns invalid JSON for <code>json_schema</code>	Warning – raw string returned in <code>response</code>

See also

- Azure AI Search built-in indexers
 - Integrated vectorization
 - How to define a skillset
 - How to generate chat completions with Azure AI model inference (Foundry)
 - Structured outputs in Azure OpenAI
-

Last updated on 11/18/2025

AML skill

Important

Support for indexer connections to the model catalog is in public preview under [supplemental terms of use](#). Preview REST APIs support this capability.

Use the AML skill to extend AI enrichment with a deployed base embedding model from the [Microsoft Foundry model catalog](#) or a custom [Azure Machine Learning \(AML\)](#) model. Your data is processed in the [Geo](#) where your model is deployed.

You specify the AML skill in a skillset, which then integrates your deployed model into an AI enrichment pipeline. The AML skill is useful for performing processing or inference not supported by built-in skills. Examples include generating embeddings with your own model and applying custom machine learning logic to enriched content.

For AML online endpoints, use a stable API version or an equivalent Azure SDK to call the AML skill. For connections to the model catalog, use a preview API version.

AML skill usage

Like other skills, the AML skill has inputs and outputs. The inputs are sent as a JSON object to a serverless deployment from the Foundry model catalog or an AML online endpoint. The output should include a success status code, JSON payload, and the parameters specified by your AML skill definition. Any other response is considered an error, and no enrichments are performed.

The indexer retries two times for the following HTTP status codes:

- 503 Service Unavailable
- 429 Too Many Requests

AML skill for models in Foundry

Azure AI Search provides the [Microsoft Foundry model catalog vectorizer](#), which is also available in the [Import data \(new\) wizard](#), for query-time connections to the model catalog. If you want to use this vectorizer for queries, the AML skill is the *indexing counterpart* for generating embeddings using a model from the model catalog.

During indexing, the AML skill can connect to the model catalog to generate vectors for the index. At query time, queries can use a vectorizer to connect to the same model to vectorize text strings. You should use the AML skill and the Microsoft Foundry model catalog vectorizer

together so that the same embedding model is used for indexing and queries. For more information, see [Use embedding models from the Foundry model catalog](#).

We recommend using the [Import data \(new\) wizard](#) to generate a skillset that includes an AML skill for deployed embedding models in Foundry. The wizard generates the AML skill definition for inputs, outputs, and mappings, providing an easy way to test a model before writing any code.

Prerequisites

- A [Foundry hub-based project](#) or an [AML workspace](#) for a custom model that you create.
- For hub-based projects only, a [serverless deployment](#) of a [supported model](#) from the Foundry model catalog.

@odata.type

Microsoft.Skills.Custom.AmlSkill

Skill parameters

Parameters are case sensitive. The parameters you use depend on what [authentication](#) your [model provider requires](#), if any.

 [Expand table](#)

Parameter name	Description
<code>uri</code>	(Required for key authentication) The target URI of the serverless deployment from the Foundry model catalog or the scoring URI of the AML online endpoint . Only the HTTPS URI scheme is allowed. Supported models from the model catalog are: <ul style="list-style-type: none">• Cohere-embed-v3-english• Cohere-embed-v3-multilingual• Cohere-embed-v4
<code>key</code>	(Required for key authentication) The API key of the model provider.
<code>resourceId</code>	(Required for token authentication) The Azure Resource Manager resource ID of the model provider. For an AML online endpoint, use the <code>subscriptions/{guid}/resourceGroups/{resource-group-}</code>

Parameter name	Description
	<code>name}/Microsoft.MachineLearningServices/workspaces/{workspace-name}/onlineendpoints/{endpoint_name}</code> format.
<code>region</code>	(Optional for token authentication) The region in which the model provider is deployed. Required if the region is different from the region of the search service.
<code>timeout</code>	(Optional) The timeout for the HTTP client making the API call. It must be formatted as an XSD "dayTimeDuration" value, which is a restricted subset of an ISO 8601 duration value. For example, <code>PT60S</code> for 60 seconds. If not set, a default value of 30 seconds is chosen. You can set the timeout to a minimum of 1 second and a maximum of 230 seconds.
<code>degreeOfParallelism</code>	(Optional) The number of calls the indexer makes in parallel to the endpoint you provide. You can decrease this value if your endpoint is failing under too high of a request load. You can raise it if your endpoint is able to accept more requests and you would like an increase in the performance of the indexer. If not set, a default value of 5 is used. You can set <code>degreeOfParallelism</code> to a minimum of 1 and a maximum of 10.

Authentication

The AML skill provides two authentication options:

- **Key-based authentication.** You provide a static key to authenticate scoring requests from the AML skill. Set the `uri` and `key` parameters for this connection.
- **Token-based authentication.** The Foundry hub-based project or AML online endpoint is deployed using token-based authentication. The Azure AI Search service must have a [managed identity](#) and a role assignment on the model provider. The AML skill then uses the search service identity to authenticate against the model provider, with no static keys required. The search service identity must have the **Owner** or **Contributor** role. Set the `resourceId` parameter, and if the search service is in a different region from the model provider, set the `region` parameter.

Skill inputs

Skill inputs are a node of the [enriched document](#) created during *document cracking*. For example, it might be the root document, a normalized image, or the content of a blob. There are no predefined inputs for this skill. For inputs, you should specify one or more nodes that are populated at the time of the AML skill's execution.

Skill outputs

Skill outputs are new nodes of an enriched document created by the skill. There are no predefined outputs for this skill. For outputs, you should provide nodes that can be populated from the JSON response of your AML skill.

Sample definition

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",  
    "description": "A custom model that detects the language in a document.",  
    "uri": "https://language-model.models.contoso.com/score",  
    "context": "/document",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "detected_language_code"  
        }  
    ]  
}
```

Sample input JSON structure

This JSON structure represents the payload sent to your Foundry hub-based project or AML online endpoint. The top-level fields of the structure correspond to the "names" specified in the `inputs` section of the skill definition. The values of those fields are from the "sources" of those fields, which could be from a field in the document or another skill.

JSON

```
{  
    "text": "Este es un contrato en Inglés"  
}
```

Sample output JSON structure

The output corresponds to the response from your Foundry hub-based project or AML online endpoint. The model provider should only return a JSON payload (verified by looking at the `Content-Type` response header) and should be an object whose fields are enrichments matching the "names" in the `output` and whose value is considered the enrichment.

JSON

```
{  
    "detected_language_code": "es"  
}
```

Inline shaping sample definition

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",  
    "description": "A sample model that detects the language of sentence",  
    "uri": "https://language-model.models.contoso.com/score",  
    "context": "/document",  
    "inputs": [  
        {  
            "name": "shapedText",  
            "sourceContext": "/document",  
            "inputs": [  
                {  
                    "name": "content",  
                    "source": "/document/content"  
                }  
            ]  
        }  
    ],  
    "outputs": [  
        {  
            "name": "detected_language_code"  
        }  
    ]  
}
```

Inline shaping input JSON structure

JSON

```
{  
    "shapedText": { "content": "Este es un contrato en Inglés" }  
}
```

Inline shaping sample output JSON structure

JSON

```
{  
    "detected_language_code": "es"  
}
```

Error cases

In addition to your Foundry hub-based project or AML online endpoint being unavailable or sending nonsuccessful status codes, the following cases are considered errors:

- The model provider returns a success status code, but the response indicates that it isn't `application/json`. The response is thus invalid, and no enrichments are performed.
- The model provider returns invalid JSON.

If the model provider is unavailable or returns an HTTP error, a friendly error with any available details about the HTTP error is added to the indexer execution history.

See also

- [Create a skillset in Azure AI Search](#)
- [Use embedding models from Foundry model catalog](#)
- [Troubleshoot Azure Machine Learning online endpoint deployment and scoring](#)

Last updated on 11/18/2025

Custom Entity Lookup cognitive skill

Article • 08/28/2024

The **Custom Entity Lookup** skill is used to detect or recognize entities that you define. During skillset execution, the skill looks for text from a custom, user-defined list of words and phrases. The skill uses this list to label any matching entities found within source documents. The skill also supports a degree of fuzzy matching that can be applied to find matches that are similar but not exact.

ⓘ Note

This skill isn't bound to an Azure AI services API but requires an Azure AI services key to allow more than 20 transactions. This skill is [metered by Azure AI Search](#).

@odata.type

Microsoft.Skills.Text.CustomEntityLookupSkill

Data limits

- The maximum input record size supported is 256 MB. If you need to break up your data before sending it to the custom entity lookup skill, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.
- The maximum size of the custom entity definition is 10 MB if it's provided as an external file, specified through the "entitiesDefinitionUri" parameter.
- If the entities are defined inline using the "inlineEntitiesDefinition" parameter, the maximum size is 10 KB.

Skill parameters

Parameters are case-sensitive.

[] Expand table

Parameter name	Description
entitiesDefinitionUri	Path to an external JSON or CSV file containing all the target text to match against. This entity definition is read at the

Parameter name	Description
	beginning of an indexer run; any updates to this file mid-run won't be realized until subsequent runs. This file must be accessible over HTTPS. See Custom Entity Definition Format below for expected CSV or JSON schema.
<code>inlineEntitiesDefinition</code>	Inline JSON entity definitions. This parameter supersedes the <code>entitiesDefinitionUri</code> parameter if present. No more than 10 KB of configuration may be provided inline. See Custom Entity Definition below for expected JSON schema.
<code>defaultLanguageCode</code>	(Optional) Language code of the input text used to tokenize and delineate input text. The following languages are supported: <code>da</code> , <code>de</code> , <code>en</code> , <code>es</code> , <code>fi</code> , <code>fr</code> , <code>it</code> , <code>pt</code> . The default is English (<code>en</code>). If you pass a <code>languagecode-countrycode</code> format, only the <code>languagecode</code> part of the format is used.
<code>globalDefaultCaseSensitive</code>	(Optional) Default case sensitive value for the skill. If <code>defaultCaseSensitive</code> value of an entity isn't specified, this value will become the <code>defaultCaseSensitive</code> value for that entity.
<code>globalDefaultAccentSensitive</code>	(Optional) Default accent sensitive value for the skill. If <code>defaultAccentSensitive</code> value of an entity isn't specified, this value will become the <code>defaultAccentSensitive</code> value for that entity.
<code>globalDefaultFuzzyEditDistance</code>	(Optional) Default fuzzy edit distance value for the skill. If <code>defaultFuzzyEditDistance</code> value of an entity isn't specified, this value will become the <code>defaultFuzzyEditDistance</code> value for that entity.

Skill inputs

[] Expand table

Input name	Description
<code>text</code>	The text to analyze.
<code>languageCode</code>	Optional. Default is <code>"en"</code> .

Skill outputs

[] Expand table

Output name	Description
<code>entities</code>	<p>An array of complex types that contains the following fields:</p> <ul style="list-style-type: none"> • <code>"name"</code>: The top-level entity; it represents the "normalized" form. • <code>"id"</code>: A unique identifier for the entity as defined in the "Custom Entity Definition". • <code>"description"</code>: Entity description as defined by the user in the "Custom Entity Definition Format". • <code>"type"</code>: Entity type as defined by the user in the "Custom Entity Definition Format". • <code>"subtype"</code>: Entity subtype as defined by the user in the "Custom Entity Definition Format". • <code>"matches"</code>: An array of complex types that contain: <ul style="list-style-type: none"> ◦ <code>"text"</code> from the source document ◦ <code>"offset"</code> location where the match was found, ◦ <code>"length"</code> of the text measured in characters ◦ <code>"matchDistance"</code> or the number of characters that differ between the match and the entity <code>"name"</code>.

Custom entity definition format

There are three approaches for providing the list of custom entities to the Custom Entity Lookup skill:

- .CSV file (UTF-8 encoded)
- .JSON file (UTF-8 encoded)
- Inline within the skill definition

If the definition file is in a .CSV or .JSON file, provide the full path in the `"entitiesDefinitionUri"` parameter. The file is downloaded at the start of each indexer run. It must remain accessible until the indexer stops.

If you're using an inline definition, specify it under the `"inlineEntitiesDefinition"` skill parameter.

Note

Indexers support specialized parsing modes for JSON and CSV files. When using the custom entity lookup skill, keep `"parsingMode"` set to `"default"`. The skill expects JSON and CSV in an unparsed state.

CSV format

You can provide the definition of the custom entities to look for in a Comma-Separated Value (CSV) file by providing the path to the file and setting it in the "entitiesDefinitionUri" skill parameter. The path should be at an https location. The definition file can be up to 10 MB in size.

The CSV format is simple. Each line represents a unique entity, as shown below:

```
Bill Gates, BillG, William H. Gates  
Microsoft, MSFT  
Satya Nadella
```

In this case, there are three entities that can be returned (Bill Gates, Satya Nadella, Microsoft). Aliases follow after the main entity. A match on an alias is bundled under the primary entity. For example, if the string "William H. Gates" is found in a document, a match for the "Bill Gates" entity will be returned.

JSON format

You can provide the definition of the custom entities to look for in a JSON file as well. The JSON format gives you a bit more flexibility since it allows you to define matching rules per term. For instance, you can specify the fuzzy matching distance (Damerau-Levenshtein distance) for each term or whether the matching should be case-sensitive or not.

Just like with CSV files, you need to provide the path to the JSON file and set it in the "entitiesDefinitionUri" skill parameter. The path should be at an https location. The definition file can be up to 10 MB in size.

The most basic JSON custom entity list definition can be a list of entities to match:

```
JSON  
[  
  {  
    "name" : "Bill Gates"  
  },  
  {  
    "name" : "Microsoft"  
  },  
  {  
    "name" : "Satya Nadella"  
  }
```

```
        }  
    ]
```

More complex definitions can provide a user-defined ID, description, type, subtype, and aliases. If an alias term is matched, the entity will be returned as well:

JSON

```
[  
  {  
    "name" : "Bill Gates",  
    "description" : "Microsoft founder.",  
    "aliases" : [  
      { "text" : "William H. Gates", "caseSensitive" : false },  
      { "text" : "BillG", "caseSensitive" : true }  
    ]  
  },  
  {  
    "name" : "Xbox One",  
    "type": "Hardware",  
    "subtype" : "Gaming Device",  
    "id" : "4e36bf9d-5550-4396-8647-8e43d7564a76",  
    "description" : "The Xbox One product"  
  },  
  {  
    "name" : "LinkedIn" ,  
    "description" : "The LinkedIn company",  
    "id" : "differentIdentifyingScheme123",  
    "fuzzyEditDistance" : 0  
  },  
  {  
    "name" : "Microsoft" ,  
    "description" : "Microsoft Corporation",  
    "id" : "differentIdentifyingScheme987",  
    "defaultCaseSensitive" : false,  
    "defaultFuzzyEditDistance" : 1,  
    "aliases" : [  
      { "text" : "MSFT", "caseSensitive" : true }  
    ]  
  }  
]
```

The tables below describe the configuration parameters you can set when defining custom entities:

[\[\] Expand table](#)

Field name	Description
name	The top-level entity descriptor. Matches in the skill output will be grouped by this name, and it should represent the "normalized" form of the text being found.
description	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
type	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
subtype	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
id	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
caseSensitive	(Optional) Defaults to false. Boolean value denoting whether comparisons with the entity name should be sensitive to character casing. Sample case insensitive matches of "Microsoft" could be: microsoft, microSoft, MICROSOFT
accentSensitive	(Optional) Defaults to false. Boolean value denoting whether accented and unaccented letters such as 'é' and 'e' should be identical.
fuzzyEditDistance	(Optional) Defaults to 0. Maximum value of 5. Denotes the acceptable number of divergent characters that would still constitute a match with the entity name. The smallest possible fuzziness for any given match is returned. For instance, if the edit distance is set to 3, "Windows 10" would still match "Windows", "Windows10" and "windows 7". When case sensitivity is set to false, case differences do NOT count towards fuzziness tolerance, but otherwise do.
defaultCaseSensitive	(Optional) Changes the default case sensitivity value for this entity. It can be used to change the default value of all aliases caseSensitive values.
defaultAccentSensitive	(Optional) Changes the default accent sensitivity value for this entity. It can be used to change the default value of all aliases accentSensitive values.
defaultFuzzyEditDistance	(Optional) Changes the default fuzzy edit distance value for this entity. It can be used to change the default value of all aliases fuzzyEditDistance values.

Field name	Description
aliases	(Optional) An array of complex objects that can be used to specify alternative spellings or synonyms to the root entity name.

[\[\] Expand table](#)

Alias properties	Description
text	The alternative spelling or representation of some target entity name.
caseSensitive	(Optional) Acts the same as root entity "caseSensitive" parameter above, but applies to only this one alias.
accentSensitive	(Optional) Acts the same as root entity "accentSensitive" parameter above, but applies to only this one alias.
fuzzyEditDistance	(Optional) Acts the same as root entity "fuzzyEditDistance" parameter above, but applies to only this one alias.

Inline format

In some cases, it may be more convenient to embed the custom entity definition so that its inline with the skill definition. You can use the same JSON format as the one described above, except that it's included within the skill definition. Only configurations that are less than 10 KB in size (serialized size) can be defined inline.

Sample skill definition

A sample skill definition using an inline format is shown below:

```
JSON

{
  "@odata.type": "#Microsoft.Skills.Text.CustomEntityLookupSkill",
  "context": "/document",
  "inlineEntitiesDefinition": [
    {
      "name" : "Bill Gates",
      "description" : "Microsoft founder." ,
      "aliases" : [
        { "text" : "William H. Gates", "caseSensitive" : false },
        { "text" : "BillG", "caseSensitive" : true }
      ]
    },
    {
      "name" : "Steve Jobs",
      "description" : "Apple founder." ,
      "aliases" : [
        { "text" : "Stephen Jobs", "caseSensitive" : false },
        { "text" : "SteveJ", "caseSensitive" : true }
      ]
    }
  ]
}
```

```

        "name" : "Xbox One",
        "type": "Hardware",
        "subtype" : "Gaming Device",
        "id" : "4e36bf9d-5550-4396-8647-8e43d7564a76",
        "description" : "The Xbox One product"
    }
],
"inputs": [
{
    "name": "text",
    "source": "/document/content"
}
],
"outputs": [
{
    "name": "entities",
    "targetName": "matchedEntities"
}
]
}

```

Alternatively, you can point to an external entities definition file. A sample skill definition using the `entitiesDefinitionUri` format is shown below:

JSON

```
{
"@odata.type": "#Microsoft.Skills.Text.CustomEntityLookupSkill",
"context": "/document",
"entitiesDefinitionUri": "https://myblobhost.net/keyWordsConfig.csv",
"inputs": [
{
    "name": "text",
    "source": "/document/content"
}
],
"outputs": [
{
    "name": "entities",
    "targetName": "matchedEntities"
}
]
}
```

Sample index definition

This section provides a sample index definition. Both "entities" and "matches" are arrays of complex types. You can have multiple entities per document, and multiple matches for each entity.

JSON

```
{  
  "name": "entities",  
  "type": "Collection(Edm.ComplexType)",  
  "fields": [  
    {  
      "name": "name",  
      "type": "Edm.String",  
      "facetable": false,  
      "filterable": false,  
      "retrievable": true,  
      "searchable": true,  
      "sortable": false,  
    },  
    {  
      "name": "id",  
      "type": "Edm.String",  
      "facetable": false,  
      "filterable": false,  
      "retrievable": true,  
      "searchable": false,  
      "sortable": false,  
    },  
    {  
      "name": "description",  
      "type": "Edm.String",  
      "facetable": false,  
      "filterable": false,  
      "retrievable": true,  
      "searchable": true,  
      "sortable": false,  
    },  
    {  
      "name": "type",  
      "type": "Edm.String",  
      "facetable": true,  
      "filterable": true,  
      "retrievable": true,  
      "searchable": false,  
      "sortable": false,  
    },  
    {  
      "name": "subtype",  
      "type": "Edm.String",  
      "facetable": true,  
      "filterable": true,  
      "retrievable": true,  
      "searchable": false,  
      "sortable": false,  
    },  
    {  
      "name": "matches",  
      "type": "Collection(Edm.ComplexType)",  
    }  
  ]  
}
```

```

"fields": [
  {
    "name": "text",
    "type": "Edm.String",
    "facetable": false,
    "filterable": false,
    "retrievable": true,
    "searchable": true,
    "sortable": false,
  },
  {
    "name": "offset",
    "type": "Edm.Int32",
    "facetable": true,
    "filterable": true,
    "retrievable": true,
    "sortable": false,
  },
  {
    "name": "length",
    "type": "Edm.Int32",
    "facetable": true,
    "filterable": true,
    "retrievable": true,
    "sortable": false,
  },
  {
    "name": "matchDistance",
    "type": "Edm.Double",
    "facetable": true,
    "filterable": true,
    "retrievable": true,
    "sortable": false,
  }
]
}
]
}

```

Sample input data

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "The company, Microsoft, was founded by Bill Gates.  
Microsoft's gaming console is called Xbox",
        "languageCode": "en"
      }
    }
  ]
}
```

```
        }
    ]
}
```

Sample output

JSON

```
{
  "values" :
  [
    {
      "recordId": "1",
      "data" : {
        "entities": [
          {
            "name" : "Microsoft",
            "description" : "This document refers to Microsoft the company",
            "id" : "differentIdentifyingScheme987",
            "matches" : [
              {
                "text" : "microsoft",
                "offset" : 13,
                "length" : 9,
                "matchDistance" : 0
              },
              {
                "text" : "Microsoft",
                "offset" : 49,
                "length" : 9,
                "matchDistance" : 0
              }
            ]
          },
          {
            "name" : "Bill Gates",
            "description" : "William Henry Gates III, founder of Microsoft.",
            "matches" : [
              {
                "text" : "Bill Gates",
                "offset" : 37,
                "length" : 10,
                "matchDistance" : 0
              }
            ]
          }
        ]
      }
    }
}
```

```
]  
}
```

Warnings

```
"Reached maximum capacity for matches, skipping all further duplicate matches."
```

This warning will be emitted if the number of matches detected is greater than the maximum allowed. No more duplicate matches will be returned. If you need a higher threshold, you can file a [support ticket](#) for assistance with your individual use case.

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Entity Recognition skill \(to search for well known entities\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Custom Web API skill in an Azure AI Search enrichment pipeline

Article • 04/14/2025

The **Custom Web API** skill allows you to extend AI enrichment by calling out to a Web API endpoint providing custom operations. Similar to built-in skills, a **Custom Web API** skill has inputs and outputs. Depending on the inputs, your Web API receives a JSON payload when the indexer runs, and outputs a JSON payload as a response, along with a success status code. The response is expected to have the outputs specified by your custom skill. Any other response is considered an error and no enrichments are performed. The structure of the JSON payload is described further down in this document.

The **Custom Web API** skill is also used in the implementation of [Azure OpenAI On Your Data](#) feature. If Azure OpenAI is [configured for role-based access](#) and you get `403 Forbidden` calls when creating the vector index, verify that Azure AI Search has a [system assigned identity](#) and runs as a [trusted service](#) on Azure OpenAI.

ⓘ Note

The indexer retries twice for certain standard HTTP status codes returned from the Web API. These HTTP status codes are:

- `502 Bad Gateway`
- `503 Service Unavailable`
- `429 Too Many Requests`

@odata.type

Microsoft.Skills.Custom.WebApiSkill

Skill parameters

Parameters are case-sensitive.

[+] [Expand table](#)

Parameter name	Description
<code>uri</code>	The URI of the Web API to which the JSON payload is sent. Only the <code>https</code> URI scheme is allowed.

Parameter name	Description
<code>authResourceId</code>	(Optional) A string that if set, indicates that this skill should use a system managed identity on the connection to the function or app hosting the code. This property takes an application (client) ID or app's registration in Microsoft Entra ID, in any of these formats: <code>api://<appId></code> , <code><appId>/default</code> , <code>api://<appId>/default</code> . This value is used to scope the authentication token retrieved by the indexer, and is sent along with the custom Web skill API request to the function or app. Setting this property requires that your search service is configured for managed identity and your Azure function app is configured for a Microsoft Entra sign in . To use this parameter, call the API with <code>api-version=2023-10-01-Preview</code> .
<code>authIdentity</code>	(Optional) A user-managed identity used by the search service for connecting to the function or app hosting the code. You can use either a system or user managed identity . To use a system manged identity, leave <code>authIdentity</code> blank.
<code>httpMethod</code>	The method to use while sending the payload. Allowed methods are <code>PUT</code> or <code>POST</code>
<code>httpHeaders</code>	A collection of key-value pairs where the keys represent header names and values represent header values that are sent to your Web API along with the payload. The following headers are prohibited from being in this collection: <code>Accept</code> , <code>Accept-Charset</code> , <code>Accept-Encoding</code> , <code>Content-Length</code> , <code>Content-Type</code> , <code>Cookie</code> , <code>Host</code> , <code>TE</code> , <code>Upgrade</code> , <code>Via</code> .
<code>timeout</code>	(Optional) When specified, indicates the timeout for the http client making the API call. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). For example, <code>PT60S</code> for 60 seconds. If not set, a default value of 30 seconds is chosen. The timeout can be set to a maximum of 230 seconds and a minimum of 1 second.
<code>batchSize</code>	(Optional) Indicates how many "data records" (see JSON payload structure below) is sent per API call. If not set, a default of 1000 is chosen. We recommend that you make use of this parameter to achieve a suitable tradeoff between indexing throughput and load on your API.
<code>degreeOfParallelism</code>	(Optional) When specified, indicates the number of calls the indexer makes in parallel to the endpoint you provide. You can decrease this value if your endpoint is failing under pressure, or raise it if your endpoint can handle the load. If not set, a default value of 5 is used. The <code>degreeOfParallelism</code> can be set to a maximum of 10 and a minimum of 1.

Skill inputs

There are no predefined inputs for this skill. The inputs are any existing field, or any [node in the enrichment tree](#) that you want to pass to your custom skill.

Skill outputs

There are no predefined outputs for this skill. Be sure to [define an output field mapping](#) in the indexer if the skill's output should be sent to a field in the search index.

Sample definition

```
JSON

{
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
    "description": "A custom skill that can identify positions of different phrases in the source text",
    "uri": "https://contoso.count-things.com",
    "batchSize": 4,
    "context": "/document",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        },
        {
            "name": "language",
            "source": "/document/languageCode"
        },
        {
            "name": "phraseList",
            "source": "/document/keyphrases"
        }
    ],
    "outputs": [
        {
            "name": "hitPositions"
        }
    ]
}
```

Sample input JSON structure

This JSON structure represents the payload that is sent to your Web API. It always follows these constraints:

- The top-level entity is called `values` and is an array of objects. The number of such objects are at most the `batchSize`.
- Each object in the `values` array has:
 - A `recordId` property that is a **unique** string, used to identify that record.

- A `data` property that is a JSON object. The fields of the `data` property correspond to the "names" specified in the `inputs` section of the skill definition. The values of those fields are from the `source` of those fields (which could be from a field in the document, or potentially from another skill).

JSON

```
{
  "values": [
    {
      "recordId": "0",
      "data": {
        "text": "Este es un contrato en Inglés",
        "language": "es",
        "phraseList": ["Este", "Inglés"]
      }
    },
    {
      "recordId": "1",
      "data": {
        "text": "Hello world",
        "language": "en",
        "phraseList": ["Hi"]
      }
    },
    {
      "recordId": "2",
      "data": {
        "text": "Hello world, Hi world",
        "language": "en",
        "phraseList": ["world"]
      }
    },
    {
      "recordId": "3",
      "data": {
        "text": "Test",
        "language": "es",
        "phraseList": []
      }
    }
  ]
}
```

Sample output JSON structure

The "output" corresponds to the response returned from your Web API. The Web API should only return a JSON payload (verified by looking at the `Content-Type` response header) and should satisfy the following constraints:

- There should be a top-level entity called `values`, which should be an array of objects.
- The number of objects in the array should be the same as the number of objects sent to the Web API.
- Each object should have:
 - A `recordId` property.
 - A `data` property, which is an object where the fields are enrichments matching the "names" in the `output` and whose value is considered the enrichment.
 - An `errors` property, an array listing any errors encountered that is added to the indexer execution history. This property is required, but can have a `null` value.
 - A `warnings` property, an array listing any warnings encountered that is added to the indexer execution history. This property is required, but can have a `null` value.
- The ordering of objects in the `values` in either the request or response isn't important. However, the `recordId` is used for correlation so any record in the response containing a `recordId`, which wasn't part of the original request to the Web API is discarded.

JSON

```
{  
  "values": [  
    {  
      "recordId": "3",  
      "data": {},  
      "errors": [  
        {  
          "message" : "'phraseList' should not be null or empty"  
        }  
      ],  
      "warnings": null  
    },  
    {  
      "recordId": "2",  
      "data": {  
        "hitPositions": [6, 16]  
      },  
      "errors": null,  
      "warnings": null  
    }  
  ]  
}
```

```
        },
        {
            "recordId": "0",
            "data": {
                "hitPositions": [0, 23]
            },
            "errors": null,
            "warnings": null
        },
        {
            "recordId": "1",
            "data": {
                "hitPositions": []
            },
            "errors": null,
            "warnings": [
                {
                    "message": "No occurrences of 'Hi' were found in the input text"
                }
            ]
        },
    ],
}
}
```

Error cases

In addition to your Web API being unavailable, or sending out non-successful status codes the following are considered erroneous cases:

- If the Web API returns a success status code but the response indicates that it isn't `application/json` then the response is considered invalid and no enrichments are performed.
- If there are invalid records (for example, `recordId` is missing or duplicated) in the response `values` array, no enrichment is performed for the invalid records. It's important to adhere to the Web API skill contract when developing custom skills. You can refer to [this example ↗](#) provided in the [Power Skill repository ↗](#) that follows the expected contract.

For cases when the Web API is unavailable or returns an HTTP error, a friendly error with any available details about the HTTP error is added to the indexer execution history.

See also

- [Define a skillset](#)

- Add custom skill to an AI enrichment pipeline
- Example: Creating a custom skill for AI enrichment
- Power Skill repository ↗

Conditional cognitive skill

Article • 08/28/2024

The **Conditional** skill enables Azure AI Search scenarios that require a Boolean operation to determine the data to assign to an output. These scenarios include filtering, assigning a default value, and merging data based on a condition.

The following pseudocode demonstrates what the conditional skill accomplishes:

```
if (condition)
    { output = whenTrue }
else
    { output = whenFalse }
```

ⓘ Note

This skill isn't bound to Azure AI services. It is non-billable and has no Azure AI services key requirement.

@odata.type

Microsoft.Skills.Util.ConditionalSkill

Evaluated fields

This skill is special because its inputs are evaluated fields.

The following items are valid values of an expression:

- Annotation paths (paths in expressions must be delimited by "\$(" and ")"")
Examples:

```
"= $(/document)"
 "= $(/document/content)"
```

- Literals (strings, numbers, true, false, null)
Examples:

```
"= 'this is a string'"    // string (note the single quotation marks)
 "= 34"                  // number
 "= true"                // Boolean
 "= null"                // null value
```

- Expressions that use comparison operators (==, !=, >=, >, <=, <)

Examples:

```
"= $(/document/language) == 'en'"
 "= $(/document/sentiment) >= 0.5"
```

- Expressions that use Boolean operators (&&, ||, !, ^)

Examples:

```
"= $(/document/language) == 'en' && $(/document/sentiment) > 0.5"
 "= !true"
```

- Expressions that use numeric operators (+, -, *, /, %)

Examples:

```
"= $(/document/sentiment) + 0.5"           // addition
 "= $(/document/totalValue) * 1.10"          // multiplication
 "= $(/document/lengthInMeters) / 0.3049"     // division
```

Because the conditional skill supports evaluation, you can use it in minor-transformation scenarios. For example, see [skill definition 4](#).

Skill inputs

Inputs are case-sensitive.

 Expand table

Input	Description
condition	<p>This input is an evaluated field that represents the condition to evaluate. This condition should evaluate to a Boolean value (<i>true</i> or <i>false</i>).</p> <p>Examples:</p> <pre>= true = \$(/document/language) =='fr' = \$(/document/pages/*/language) == \$(/document/expectedLanguage)"</pre>
whenTrue	<p>This input is an evaluated field that represents the value to return if the condition is evaluated to <i>true</i>. Constants strings should be returned in single quotation marks (' and ').</p> <p>Sample values:</p> <pre>= 'contract' = \$(/document/contractType)" = \$(/document/entities/*)"</pre>
whenFalse	<p>This input is an evaluated field that represents the value to return if the condition is evaluated to <i>false</i>.</p> <p>Sample values:</p> <pre>= 'contract" = \$(/document/contractType)" = \$(/document/entities/*)"</pre>

Skill outputs

There's a single output that's simply called "output." It returns the value *whenFalse* if the condition is false or *whenTrue* if the condition is true.

Examples

Sample skill definition 1: Filter documents to return only French documents

The following output returns an array of sentences ("\$/document/frenchSentences") if the language of the document is French. If the language isn't French, the value is set to *null*.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
  "context": "/document",
  "inputs": [
    { "name": "condition", "source": "= $(/document/language) == 'fr'" }
},
```

```

        { "name": "whenTrue", "source": "/document/sentences" },
        { "name": "whenFalse", "source": "= null" }
    ],
    "outputs": [ { "name": "output", "targetName": "frenchSentences" } ]
}

```

If "/document/frenchSentences" is used as the *context* of another skill, that skill only runs if "/document/frenchSentences" isn't set to *null*.

Sample skill definition 2: Set a default value for a value that doesn't exist

The following output creates an annotation (""/document/languageWithDefault") that's set to the language of the document or to "es" if the language isn't set.

JSON

```

{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document",
    "inputs": [
        { "name": "condition", "source": "= $(/document/language) == null" }
    ],
    { "name": "whenTrue", "source": "= 'es'" },
    { "name": "whenFalse", "source": "= $(/document/language)" }
],
    "outputs": [ { "name": "output", "targetName": "languageWithDefault" } ]
}

```

Sample skill definition 3: Merge values from two fields into one

In this example, some sentences have a *frenchSentiment* property. Whenever the *frenchSentiment* property is null, we want to use the *englishSentiment* value. We assign the output to a member that's called *sentiment* (""/document/sentences/*/sentiment").

JSON

```

{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document/sentences/*",
    "inputs": [
        { "name": "condition", "source": "= $(/document/sentences/*/frenchSentiment) == null" },
        { "name": "whenTrue", "source": "/document/sentences/*/englishSentiment" },

```

```
        { "name": "whenFalse", "source":  
  "/document/sentences/*/frenchSentiment" }  
    ],  
    "outputs": [ { "name": "output", "targetName": "sentiment" } ]  
}
```

Transformation example

Sample skill definition 4: Data transformation on a single field

In this example, we receive a *sentiment* that's between 0 and 1. We want to transform it to be between -1 and 1. We can use the conditional skill to do this minor transformation.

In this example, we don't use the conditional aspect of the skill because the condition is always *true*.

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",  
  "context": "/document/sentences/*",  
  "inputs": [  
    { "name": "condition", "source": "= true" },  
    { "name": "whenTrue", "source": "=  
$(/document/sentences/*/sentiment) * 2 - 1" },  
    { "name": "whenFalse", "source": "= 0" }  
  ],  
  "outputs": [ { "name": "output", "targetName": "normalizedSentiment" } ]  
}
```

Special considerations

Some parameters are evaluated, so you need to be especially careful to follow the documented pattern. Expressions must start with an equals sign. A path must be delimited by "\$(" and ")". Make sure to put strings in single quotation marks. That helps the evaluator distinguish between strings and actual paths and operators. Also, make sure to put white space around operators (for instance, a "*" in a path means something different than multiply).

Next steps

- Built-in skills
 - How to define a skillset
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Document Extraction cognitive skill

07/29/2025

The **Document Extraction** skill extracts content from a file within the enrichment pipeline. By default, content extraction or retrieval is built into the indexer pipeline. However, by using the Document Extraction skill, you can control how parameters are set, and how extracted content is named in the enrichment tree.

For [vector](#) and [multimodal search](#), Document Extraction combined with the [Text Split skill](#) is more affordable than other [data chunking approaches](#). The following tutorials demonstrate skill usage for different scenarios:

- [Tutorial: Vectorize images and text](#)
- [Tutorial: Verbalize images using generative AI](#)

!**Note**

This skill isn't bound to Azure AI services and has no Azure AI services key requirement.

This skill extracts text and images. Text extraction is free. Image extraction is [billable by Azure AI Search](#). On a free search service, the cost of 20 transactions per indexer per day is absorbed so that you can complete quickstarts, tutorials, and small projects at no charge. For basic and higher tiers, image extraction is billable.

@odata.type

Microsoft.Skills.Util.DocumentExtractionSkill

Supported document formats

The DocumentExtractionSkill can extract text from the following document formats:

- CSV (see [Indexing CSV blobs](#))
- EML
- EPUB
- GZ
- HTML
- JSON (see [Indexing JSON blobs](#))
- KML (XML for geographic representations)

- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also [Indexing plain text](#))
- RTF
- XML
- ZIP

Skill parameters

Parameters are case-sensitive.

 [Expand table](#)

Inputs	Allowed Values	Description
<code>parsingMode</code>	<code>default</code> <code>text</code> <code>json</code>	<p>Set to <code>default</code> for document extraction from files that aren't pure text or json. For source files that contain mark up (such as PDF, HTML, RTF, and Microsoft Office files), use the default to extract just the text, minus any markup language or tags. If <code>parsingMode</code> isn't defined explicitly, it will be set to <code>default</code>.</p> <p>Set to <code>text</code> if source files are TXT. This parsing mode improves performance on plain text files. If files include markup, this mode will preserve the tags in the final output.</p> <p>Set to <code>json</code> to extract structured content from json files.</p>
<code>dataToExtract</code>	<code>contentAndMetadata</code> <code>allMetadata</code>	<p>Set to <code>contentAndMetadata</code> to extract all metadata and textual content from each file. If <code>dataToExtract</code> isn't defined explicitly, it will be set to <code>contentAndMetadata</code>.</p> <p>Set to <code>allMetadata</code> to extract only the metadata properties for the content type (for example, metadata unique to just .png files).</p>
<code>configuration</code>	See below.	A dictionary of optional parameters that adjust how the document extraction is performed. See the below table for descriptions of supported configuration properties.

 [Expand table](#)

Configuration Parameter	Allowed Values	Description
<code>imageAction</code>	<code>none</code> <code>generateNormalizedImages</code> <code>generateNormalizedImagePerPage</code>	<p>Set to <code>none</code> to ignore embedded images or image files in the data set, or if the source data doesn't include image files. This is the default.</p> <p>For OCR and image analysis, set to <code>generateNormalizedImages</code> to have the skill create an array of normalized images as part of document cracking. This action requires that <code>parsingMode</code> is set to <code>default</code> and <code>dataToExtract</code> is set to <code>contentAndMetadata</code>. A normalized image refers to extra processing resulting in uniform image output, sized and rotated to promote consistent rendering when you include images in visual search results (for example, same-size photographs in a graph control as seen in the JFK demo). This information is generated for each image when you use this option.</p> <p>If you set to <code>generateNormalizedImagePerPage</code>, PDF files are treated differently in that instead of extracting embedded images, each page is rendered as an image and normalized accordingly. Non-PDF file types are treated the same as if <code>generateNormalizedImages</code> was set.</p>
<code>normalizedImageMaxWidth</code>	Any integer between 50-10000	The maximum width (in pixels) for normalized images generated. The default is 2000.
<code>normalizedImageMaxHeight</code>	Any integer between 50-10000	The maximum height (in pixels) for normalized images generated. The default is 2000.

(!) Note

The default of 2000 pixels for the normalized images maximum width and height is based on the maximum sizes supported by the [OCR skill](#) and the [image analysis skill](#). The [OCR skill](#) supports a maximum width and height of 4200 for non-English languages, and 10000

for English. If you increase the maximum limits, processing could fail on larger images depending on your skillset definition and the language of the documents.

Skill inputs

[Expand table](#)

Input name	Description
file_data	The file that content should be extracted from.

The "file_data" input must be an object defined as:

JSON

```
{  
  "$type": "file",  
  "data": "BASE64 encoded string of the file"  
}
```

Alternatively, it can be defined as:

JSON

```
{  
  "$type": "file",  
  "url": "URL to download file",  
  "sasToken": "OPTIONAL: SAS token for authentication if the URL provided is for a  
file in blob storage"  
}
```

The file reference object can be generated one of three ways:

- Setting the `allowSkillsetToReadFileData` parameter on your indexer definition to "true". This creates a path `/document/file_data` that is an object representing the original file data downloaded from your blob data source. This parameter only applies to files in Blob storage.
- Setting the `imageAction` parameter on your indexer definition to a value other than `none`. This creates an array of images that follows the required convention for input to this skill if passed individually (that is, `/document/normalized_images/*`).

- Having a custom skill return a json object defined EXACTLY as above. The `$type` parameter must be set to exactly `file` and the `data` parameter must be the base 64 encoded byte array data of the file content, or the `url` parameter must be a correctly formatted URL with access to download the file at that location.

Skill outputs

[] [Expand table](#)

Output name	Description
<code>content</code>	The textual content of the document.
<code>normalized_images</code>	When the <code>imageAction</code> is set to a value other than <code>none</code> , the new <code>normalized_images</code> field contains an array of images. See Extract text and information from images for more details on the output format.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.DocumentExtractionSkill",
  "parsingMode": "default",
  "dataToExtract": "contentAndMetadata",
  "configuration": {
    "imageAction": "generateNormalizedImages",
    "normalizedImageMaxWidth": 2000,
    "normalizedImageMaxHeight": 2000
  },
  "context": "/document",
  "inputs": [
    {
      "name": "file_data",
      "source": "/document/file_data"
    }
  ],
  "outputs": [
    {
      "name": "content",
      "targetName": "extracted_content"
    },
    {
      "name": "normalized_images",
      "targetName": "extracted_normalized_images"
    }
  ]
}
```

```
    ]  
}
```

Sample input

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "file_data": {  
            "$type": "file",  
            "data": "aGVsbG8="  
          }  
        }  
    }  
  ]  
}
```

Sample output

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data": {  
        "content": "hello",  
        "normalized_images": []  
      }  
    }  
  ]  
}
```

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [How to process and extract information from images](#)

Shaper cognitive skill

Article • 08/28/2024

The **Shaper** skill is used to reshape or modify the structure of the [in-memory enrichment tree](#) created by a skillset. If skill outputs can't be mapped directly to search fields, you can add a **Shaper** skill to create the data shape you need for your search index or knowledge store.

Primary use-cases for this skill include:

- You're populating a knowledge store. The physical structure of the tables and objects of a knowledge store are defined through projections. A **Shaper** skill adds granularity by creating data shapes that can be pushed to the projections.
- You want to map multiple skill outputs into a single structure in your search index, usually a [complex type](#), as described in [scenario 1](#).
- Skills produce multiple outputs, but you want to combine into a single field (it doesn't have to be a complex type), as described in [scenario 2](#). For example, combining titles and authors into a single field.
- Skills produce multiple outputs with child elements, and you want to combine them. This use-case is illustrated in [scenario 3](#).

The output name of a **Shaper** skill is always "output". Internally, the pipeline can map a different name, such as "analyzedText" as shown in the examples below, but the **Shaper** skill itself returns "output" in the response. This might be important if you are debugging enriched documents and notice the naming discrepancy, or if you build a custom skill and are structuring the response yourself.

ⓘ Note

This skill isn't bound to Azure AI services. It is non-billable and has no Azure AI services key requirement.

@odata.type

Microsoft.Skills.Util.ShaperSkill

Scenario 1: complex types

Consider a scenario where you want to create a structure called *analyzedText* that has two members: *text* and *sentiment*, respectively. In an index, a multi-part searchable field is called a *complex type* and it's often created when source data has a corresponding complex structure that maps to it.

However, another approach for creating complex types is through the **Shaper** skill. By including this skill in a skillset, the in-memory operations during skillset processing can output data shapes with nested structures, which can then be mapped to a complex type in your index.

The following example skill definition provides the member names as the input.

JSON

```
{  
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",  
  "context": "/document/content/phrases/*",  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content/phrases/*"  
    },  
    {  
      "name": "sentiment",  
      "source": "/document/content/phrases/*/sentiment"  
    }  
  ],  
  "outputs": [  
    {  
      "name": "output",  
      "targetName": "analyzedText"  
    }  
  ]  
}
```

Sample index

A skillset is invoked by an indexer, and an indexer requires an index. A complex field representation in your index might look like the following example.

JSON

```
"name": "my-index",  
"fields": [  
  { "name": "myId", "type": "Edm.String", "key": true, "filterable": true },  
  { "name": "analyzedText", "type": "Edm.ComplexType",  
    "fields": [  
      {
```

```
        "name": "text",
        "type": "Edm.String",
        "facetable": false,
        "filterable": false,
        "searchable": true,
        "sortable": false },
    {
        "name": "sentiment",
        "type": "Edm.Double",
        "facetable": true,
        "filterable": true,
        "searchable": true,
        "sortable": true }
}
```

Skill input

An incoming JSON document providing usable input for this **Shaper** skill could be:

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "this movie is awesome",
                "sentiment": 0.9
            }
        }
    ]
}
```

Skill output

The **Shaper** skill generates a new element called *analyzedText* with the combined elements of *text* and *sentiment*. This output conforms to the index schema. It will be imported and indexed in an Azure AI Search index.

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "analyzedText":
```

```

        {
            "text": "this movie is awesome" ,
            "sentiment": 0.9
        }
    }
]
}

```

Scenario 2: input consolidation

In another example, imagine that at different stages of pipeline processing, you have extracted the title of a book, and chapter titles on different pages of the book. You could now create a single structure composed of these various outputs.

The **Shaper** skill definition for this scenario might look like the following example:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "context": "/document",
    "inputs": [
        {
            "name": "title",
            "source": "/document/content/title"
        },
        {
            "name": "chapterTitles",
            "source": "/document/content/pages/*/chapterTitles/*/title"
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "titlesAndChapters"
        }
    ]
}
```

Skill output

In this case, the **Shaper** flattens all chapter titles to create a single array.

JSON

```
{
    "values": [

```

```

{
    "recordId": "1",
    "data": {
        "titlesAndChapters": {
            "title": "How to be happy",
            "chapterTitles": [
                "Start young",
                "Laugh often",
                "Eat, sleep and exercise"
            ]
        }
    }
}
]
}

```

Scenario 3: input consolidation from nested contexts

Imagine you have chapter titles and chapter numbers of a book and have run entity recognition and key phrases on the contents and now need to aggregate results from the different skills into a single shape with the chapter name, entities, and key phrases.

This example adds an optional `sourceContext` property to the "chapterTitles" input. The `source` and `sourceContext` properties are mutually exclusive. If the input is at the context of the skill, you can use `source`. If the input is at a *different* context than the skill context, use `sourceContext`. The `sourceContext` requires you to define a nested input, where each input has a `source` that identifies the specific element used to populate the named node.

The **Shaper** skill definition for this scenario might look like the following example:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "context": "/document",
    "inputs": [
        {
            "name": "title",
            "source": "/document/content/title"
        },
        {
            "name": "chapterTitles",
            "sourceContext": "/document/content/pages/*/chapterTitles/*",
            "inputs": [
                {

```

```

        "name": "title",
        "source": "/document/content/pages/*/chapterTitles/*/title"
    },
    {
        "name": "number",
        "source": "/document/content/pages/*/chapterTitles/*/number"
    }
]
}
],
"outputs": [
{
    "name": "output",
    "targetName": "titlesAndChapters"
}
]
}

```

Skill output

In this case, the **Shaper** creates a complex type. This structure exists in-memory. If you want to save it to a [knowledge store](#), you should create a projection in your skillset that defines storage characteristics.

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "titlesAndChapters": {
          "title": "How to be happy",
          "chapterTitles": [
            { "title": "Start young", "number": 1},
            { "title": "Laugh often", "number": 2},
            { "title": "Eat, sleep and exercise", "number": 3}
          ]
        }
      }
    }
  ]
}
```

See also

- Built-in skills
 - How to define a skillset
 - How to use complex types
 - Knowledge store
 - Create a knowledge store in REST
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Text Merge cognitive skill

The Text Merge skill consolidates text from an array of strings into a single field.

! Note

This skill isn't bound to Foundry Tools. It's nonbillable and has no Foundry Tools key requirement.

@odata.type

Microsoft.Skills.Text.MergeSkill

Skill parameters

Parameters are case sensitive.

[\[+\] Expand table](#)

Parameter	Description
<code>name</code>	
<code>insertPreTag</code>	String to be included before every insertion. The default value is <code>" "</code> . To omit the space, set the value to <code>""</code> .
<code>insertPostTag</code>	String to be included after every insertion. The default value is <code>" "</code> . To omit the space, set the value to <code>""</code> .

Skill inputs

[\[+\] Expand table](#)

Input name	Description
<code>itemsToInsert</code>	Array of strings to be merged.
<code>text</code>	(optional) Main text body to be inserted into. If <code>text</code> is not provided, elements of <code>itemsToInsert</code> will be concatenated.
<code>offsets</code>	(optional) Array of positions within <code>text</code> where <code>itemsToInsert</code> should be inserted. If provided, the number of elements of <code>text</code> must equal the number of elements of <code>textToInsert</code> . Otherwise all items will be appended at the end of <code>text</code> .

Skill outputs

 Expand table

Output name	Description
mergedText	The resulting merged text.
mergedOffsets	Array of positions within mergedText where elements of itemsToInsert were inserted.

Sample input

A JSON document providing usable input for this skill could be:

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "text": "The brown fox jumps over the dog",  
          "itemsToInsert": ["quick", "lazy"],  
          "offsets": [3, 28]  
        }  
    }  
  ]  
}
```

Sample output

This example shows the output of the previous input, assuming that the *insertPreTag* is set to `" "`, and *insertPostTag* is set to `""`.

JSON

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "mergedText": "The quick brown fox jumps over the lazy dog"  
        }  
    }  
  ]  
}
```

```
]  
}
```

Extended sample skillset definition

A common scenario for using Text Merge is to merge the textual representation of images (text from an OCR skill, or the caption of an image) into the content field of a document.

The following example skillset uses the OCR skill to extract text from images embedded in the document. Next, it creates a *merged_text* field to contain both original and OCRed text from each image. You can learn more about the OCR skill [here](#).

JSON

```
{  
  "description": "Extract text from images and merge with content text to produce  
  merged_text",  
  "skills":  
  [  
    {  
      "description": "Extract text (plain and structured) from image.",  
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",  
      "context": "/document/normalized_images/*",  
      "defaultLanguageCode": "en",  
      "detectOrientation": true,  
      "inputs": [  
        {  
          "name": "image",  
          "source": "/document/normalized_images/*"  
        }  
      ],  
      "outputs": [  
        {  
          "name": "text"  
        }  
      ]  
    },  
    {  
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",  
      "description": "Create merged_text, which includes all the textual  
      representation of each image inserted at the right location in the content field.",  
      "context": "/document",  
      "insertPreTag": " ",  
      "insertPostTag": " ",  
      "inputs": [  
        {  
          "name": "text",  
          "source": "/document/content"  
        },  
        {  
          "name": "itemsToInsert",  
          "source": "/document/normalized_images/*"  
        }  
      ]  
    }  
  ]  
}
```

```
        "source": "/document/normalized_images/*/text"
    },
    {
        "name": "offsets",
        "source": "/document/normalized_images/*/contentOffset"
    }
],
{
    "outputs": [
        {
            "name": "mergedText",
            "targetName": "merged_text"
        }
    ]
}
]
```

The example above assumes that a normalized-images field exists. To get normalized-images field, set the *imageAction* configuration in your indexer definition to *generateNormalizedImages* as shown below:

JSON

```
{  
    //...rest of your indexer definition goes here ...  
    "parameters": {  
        "configuration": {  
            "dataToExtract": "contentAndMetadata",  
            "imageAction": "generateNormalizedImages"  
        }  
    }  
}
```

See also

- Built-in skills
 - How to define a skillset
 - Create Indexer (REST)

Last updated on 11/18/2025

Text split cognitive skill

09/19/2025

ⓘ Important

Some parameters are in public preview under [Supplemental Terms of Use](#). The [preview REST API](#) supports these parameters.

The **Text Split** skill breaks text into chunks of text. You can specify whether you want to break the text into sentences or into pages of a particular length. Positional metadata like offset and ordinal position are also available as outputs. This skill is useful if there are maximum text length requirements in other skills downstream, such as embedding skills that pass data chunks to embedding models on Azure OpenAI and other model providers. For more information about this scenario, see [Chunk documents for vector search](#).

Several parameters are version-specific. The skills parameter table notes the API version in which a parameter was introduced so that you know whether a [version upgrade](#) is required. To use version-specific features such as *token chunking* in **2024-09-01-preview**, you can use the Azure portal, or target a REST API version, or check an Azure SDK change log to see if it supports the feature.

The Azure portal supports most preview features and can be used to create or update a skillset. For updates to the Text Split skill, edit the skillset JSON definition to add new preview parameters.

! Note

This skill isn't bound to Azure AI services. It's non-billable and has no Azure AI services key requirement.

@odata.type

Microsoft.Skills.Text.SplitSkill

Skill Parameters

Parameters are case-sensitive.

Parameter name	Description
<code>textSplitMode</code>	Either <code>pages</code> or <code>sentences</code> . Pages have a configurable maximum length, but the skill attempts to avoid truncating a sentence so the actual length might be smaller. Sentences are a string that terminates at sentence-ending punctuation, such as a period, question mark, or exclamation point, assuming the language has sentence-ending punctuation.
<code>maximumPageLength</code>	Only applies if <code>textSplitMode</code> is set to <code>pages</code> . For <code>unit</code> set to <code>characters</code> , this parameter refers to the maximum page length in characters as measured by <code>String.Length</code> . The minimum value is 300, the maximum is 50000, and the default value is 5000. The algorithm does its best to break the text on sentence boundaries, so the size of each chunk might be slightly less than <code>maximumPageLength</code> .
	For <code>unit</code> set to <code>azureOpenAITokens</code> , the maximum page length is the token length limit of the model. For text embedding models, a general recommendation for page length is 512 tokens.
<code>defaultLanguageCode</code>	(optional) One of the following language codes: <code>am</code> , <code>bs</code> , <code>cs</code> , <code>da</code> , <code>de</code> , <code>en</code> , <code>es</code> , <code>et</code> , <code>fr</code> , <code>he</code> , <code>hi</code> , <code>hr</code> , <code>hu</code> , <code>fi</code> , <code>id</code> , <code>is</code> , <code>it</code> , <code>ja</code> , <code>ko</code> , <code>lv</code> , <code>no</code> , <code>nl</code> , <code>pl</code> , <code>pt-PT</code> , <code>pt-BR</code> , <code>ru</code> , <code>sk</code> , <code>sl</code> , <code>sr</code> , <code>sv</code> , <code>tr</code> , <code>ur</code> , <code>zh-Hans</code> . Default is English (<code>en</code>). A few things to consider: <ul style="list-style-type: none"> • Providing a language code is useful to avoid cutting a word in half for nonwhitespace languages such as Chinese, Japanese, and Korean. • If you don't know the language in advance (for example, if you're using the LanguageDetectionSkill to detect language), we recommend the <code>en</code> default.
<code>pageOverlapLength</code>	Only applies if <code>textSplitMode</code> is set to <code>pages</code> . Each page starts with this number of characters or tokens from the end of the

Parameter name	Description
	previous page. If this parameter is set to 0, there's no overlapping text on successive pages. This example includes the parameter.
<code>maximumPagesToTake</code>	Only applies if <code>textSplitMode</code> is set to <code>pages</code> . Number of pages to return. The default is 0, which means to return all pages. You should set this value if only a subset of pages are needed. This example includes the parameter.
<code>unit</code>	Only applies if <code>textSplitMode</code> is set to <code>pages</code> . Specifies whether to chunk by <code>characters</code> (default) or <code>azureOpenAITokens</code> . Setting the unit affects <code>maximumPageLength</code> and <code>pageOverlapLength</code> .
<code>azureOpenAITokenizerParameters</code>	An object providing extra parameters for the <code>azureOpenAITokens</code> unit.
	<p><code>encoderModelName</code> is a designated tokenizer used for converting text into tokens, essential for natural language processing (NLP) tasks. Different models use different tokenizers. Valid values include <code>cl100k_base</code> (default) used by GPT-35-Turbo and GPT-4. Other valid values are <code>r50k_base</code>, <code>p50k_base</code>, and <code>p50k_edit</code>. The skill implements the <code>tiktoken</code> library by way of SharpToken and <code>Microsoft.ML.Tokenizers</code> but doesn't support every encoder. For example, there's currently no support for <code>o200k_base</code> encoding used by GPT-4o.</p>
	<p><code>allowedSpecialTokens</code> defines a collection of special tokens that are permitted within the tokenization process. Special tokens are string that you want to treat uniquely, ensuring they aren't split during tokenization. For example <code>["[START]", "[END]"]</code>. If the <code>tiktoken</code> library doesn't perform tokenization as expected, either due to language-specific limitations or other unexpected behaviors, it's recommended to use text splitting instead.</p>

Skill Inputs

 [Expand table](#)

Parameter	Description
<code>name</code>	
<code>text</code>	The text to split into substring.
<code>languageCode</code>	(Optional) Language code for the document. If you don't know the language of the text inputs (for example, if you're using LanguageDetectionSkill to detect the language), you can omit this parameter. If you set <code>languageCode</code> to a language isn't in the supported list for the <code>defaultLanguageCode</code> , a warning is emitted and the text isn't split.

Skill Outputs

[\[\]](#) Expand table

Parameter name	Description
<code>textItems</code>	<p>Output is an array of substrings that were extracted. <code>textItems</code> is the default name of the output.</p> <p><code>targetName</code> is optional, but if you have multiple Text Split skills, make sure to set <code>targetName</code> so that you don't overwrite the data from the first skill with the second one. If <code>targetName</code> is set, use it in output field mappings or in downstream skills that consume the skill output, such as an embedding skill.</p>
<code>offsets</code>	<p>Output is an array of offsets that were extracted. The value at each index is an object containing the offset of the text item at that index in three encodings: UTF-8, UTF-16, and CodePoint. <code>offsets</code> is the default name of the output.</p> <p><code>targetName</code> is optional, but if you have multiple Text Split skills, make sure to set <code>targetName</code> so that you don't overwrite the data from the first skill with the second one. If <code>targetName</code> is set, use it in output field mappings or in downstream skills that consume the skill output, such as an embedding skill.</p>
<code>lengths</code>	<p>Output is an array of lengths that were extracted. The value at each index is an object containing the offset of the text item at that index in three encodings: UTF-8, UTF-16, and CodePoint. <code>lengths</code> is the default name of the output.</p> <p><code>targetName</code> is optional, but if you have multiple Text Split skills, make sure to set <code>targetName</code> so that you don't overwrite the data from the first skill with the second one. If <code>targetName</code> is set, use it in output field mappings or in downstream skills that consume the skill output, such as an embedding skill.</p>
<code>ordinalPositions</code>	<p>Output is an array of ordinal positions corresponding to the position of the text item within the source text. <code>ordinalPositions</code> is the default name of the output.</p> <p><code>targetName</code> is optional, but if you have multiple Text Split skills, make sure to set <code>targetName</code> so that you don't overwrite the data from the first skill with the second one.</p>

Parameter name	Description
	one. If <code>targetName</code> is set, use it in output field mappings or in downstream skills that consume the skill output, such as an embedding skill.

Sample definition

JSON

```
{
  "name": "SplitSkill",
  "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
  "description": "A skill that splits text into chunks",
  "context": "/document",
  "defaultLanguageCode": "en",
  "textSplitMode": "pages",
  "unit": "azureOpenAITokens",
  "azureOpenAITokenizerParameters":{
    "encoderModelName": "cl100k_base",
    "allowedSpecialTokens": [
      "[START]",
      "[END]"
    ],
  },
  "maximumPageLength": 512,
  "inputs": [
    {
      "name": "text",
      "source": "/document/text"
    },
    {
      "name": "languageCode",
      "source": "/document/language"
    }
  ],
  "outputs": [
    {
      "name": "textItems",
      "targetName": "pages"
    }
  ]
}
```

Sample input

JSON

```
{
  "values": [
```

```
{
    "recordId": "1",
    "data": {
        "text": "This is the loan application for Joe Romero, a Microsoft employee who was born in Chile and who then moved to Australia...",
        "languageCode": "en"
    }
},
{
    "recordId": "2",
    "data": {
        "text": "This is the second document, which will be broken into several pages...",
        "languageCode": "en"
    }
}
]
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "pages": [
                    "This is the loan...",
                    "In the next section, we continue..."
                ],
                "offsets": [
                    {
                        "utf8": 0,
                        "utf16": 0,
                        "codePoint": 0
                    },
                    {
                        "utf8": 146,
                        "utf16": 146,
                        "codePoint": 146
                    }
                ],
                "lengths": [
                    {
                        "utf8": 146,
                        "utf16": 146,
                        "codePoint": 146
                    },
                    {
                        "utf8": 211,
                        "utf16": 211,
                        "codePoint": 211
                    }
                ]
            }
        }
    ]
}
```

```

        "utf16": 211,
        "codePoint": 211
    }
],
"ordinalPositions" : [
    1,
    2
]
}
},
{
"recordId": "2",
"data": {
"pages": [
    "This is the second document...",
    "In the next section of the second doc..."
],
"offsets": [
    {
        "utf8": 0,
        "utf16": 0,
        "codePoint": 0
    },
    {
        "utf8": 115,
        "utf16": 115,
        "codePoint": 115
    }
],
"lengths": [
    {
        "utf8": 115,
        "utf16": 115,
        "codePoint": 115
    },
    {
        "utf8": 209,
        "utf16": 209,
        "codePoint": 209
    }
],
"ordinalPositions" : [
    1,
    2
]
}
]
}
}

```

 **Note**

This example sets `textItems` to `pages` through `targetName`. Because `targetName` is set, `pages` is the value you should use to select the output from the Text Split skill. Use `/document/pages/*` in downstream skills, indexer [output field mappings](#), [knowledge store projections](#), and [index projections](#). This example doesn't set `offsets`, `lengths`, or `ordinalPosition` to any other name, so the value you should use in downstream skills would be unchanged. `offsets` and `lengths` are complex types rather than primitives, because they contain the values for multiple encoding types. The value you should use to obtain a specific encoding, for example UTF-8, would look like this:

```
/document/offsets/*/utf8.
```

Example for chunking and vectorization

This example is for integrated vectorization.

- `pageOverlapLength`: Overlapping text is useful in [data chunking](#) scenarios because it preserves continuity between chunks generated from the same document.
- `maximumPagesToTake`: Limits on page intake are useful in [vectorization](#) scenarios because it helps you stay under the maximum input limits of the embedding models providing the vectorization.

Sample definition

This definition adds `pageOverlapLength` of 100 characters and `maximumPagesToTake` of one.

Assuming the `maximumPageLength` is 5,000 characters (the default), then `"maximumPagesToTake": 1` processes the first 5,000 characters of each source document.

This example sets `textItems` to `myPages` through `targetName`. Because `targetName` is set, `myPages` is the value you should use to select the output from the Text Split skill. Use `/document/myPages/*` in downstream skills, indexer [output field mappings](#), [knowledge store projections](#), and [index projections](#).

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.SplitSkill",  
    "textSplitMode" : "pages",  
    "maximumPageLength": 1000,  
    "pageOverlapLength": 100,  
    "maximumPagesToTake": 1,  
    "defaultLanguageCode": "en",
```

```

"inputs": [
    {
        "name": "text",
        "source": "/document/content"
    },
    {
        "name": "languageCode",
        "source": "/document/language"
    }
],
"outputs": [
    {
        "name": "textItems",
        "targetName": "myPages"
    }
]
}

```

Sample input (same as previous example)

JSON

```

{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "This is the loan application for Joe Romero, a Microsoft employee who was born in Chile and who then moved to Australia...",
                "languageCode": "en"
            }
        },
        {
            "recordId": "2",
            "data": {
                "text": "This is the second document, which will be broken into several sections...",
                "languageCode": "en"
            }
        }
    ]
}

```

Sample output (notice the overlap)

Within each "textItems" array, trailing text from the first item is copied into the beginning of the second item.

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "myPages": [
          "This is the loan...Here is the overlap part",
          "Here is the overlap part...In the next section, we
continue..."
        ]
      }
    },
    {
      "recordId": "2",
      "data": {
        "myPages": [
          "This is the second document...Here is the overlap part...",
          "Here is the overlap part...In the next section of the second
doc..."
        ]
      }
    }
  ]
}
```

Error cases

If a language isn't supported, a warning is generated.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Deprecated Cognitive Skills in Azure AI Search

Article • 08/28/2024

This document describes cognitive skills that are considered deprecated (retired). Use the following guide for the contents:

- Skill Name: The name of the skill that will be deprecated; it maps to the @odata.type attribute.
- Last available api version: The last version of the Azure AI Search public API through which skillsets containing the corresponding deprecated skill can be created/updated. Indexers with attached skillsets with these skills will continue to run even in future API versions until the "End of support" date, at which point they start failing.
- End of support: The day after which the corresponding skill is considered unsupported and stops working. Previously created skillsets should still continue to function, but users are recommended to migrate away from a deprecated skill.
- Recommendations: Migration path forward to use a supported skill. Users are advised to follow the recommendations to continue to receive support.

If you're using the [Microsoft.Skills.Text.EntityRecognitionSkill](#) (Entity Recognition cognitive skill (v2)), this article helps you upgrade your skillset to use the [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#) which is generally available and introduces new features.

If you're using the [Microsoft.Skills.Text.SentimentSkill](#) (Sentiment cognitive skill (v2)), this article helps you upgrade your skillset to use the [Microsoft.Skills.Text.V3.SentimentSkill](#) which is generally available and introduces new features.

If you're using the [Microsoft.Skills.Text.NamedEntityRecognitionSkill](#) (Named Entity Recognition cognitive skill (v2)), this article helps you upgrade your skillset to use the [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#) which is generally available and introduces new features.

Microsoft.Skills.Text.EntityRecognitionSkill

Last available api version

2021-04-30-Preview

End of support

August 31, 2024

Recommendations

Use [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#) instead. It provides most of the functionality of the EntityRecognitionSkill at a higher quality. It also has richer information in its complex output fields.

To migrate to the [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#), make one or more of the following changes to your skill definition. You can update the skill definition using the [Update Skillset API](#).

1. *(Required)* Change the `@odata.type` from

```
"#Microsoft.Skills.Text.EntityRecognitionSkill" to  
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill".
```

2. *(Optional)* The `includeTypelessEntities` parameter is no longer supported as the new skill only returns entities with known types, so if your previous skill definition referenced it, it should now be removed.

3. *(Optional)* If you're making use of the `namedEntities` output, there are a few minor changes to the property names.

- a. `value` is renamed to `text`
- b. `confidence` is renamed to `confidenceScore`

If you need to generate the exact same property names, add a [ShaperSkill](#) to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",  
    "name": "NamedEntitiesShaper",  
    "description": "NamedEntitiesShaper",  
    "context": "/document/namedEntitiesV3",  
    "inputs": [  
        {  
            "name": "old_format",  
            "sourceContext": "/document/namedEntitiesV3/*",  
            "inputs": [  
                {  
                    "name": "value",  
                    "source": "/document/namedEntitiesV3/*/text"  
                }  
            ]  
        }  
    ]  
}
```

```

        },
        {
            "name": "offset",
            "source": "/document/namedEntitiesV3/*/offset"
        },
        {
            "name": "category",
            "source": "/document/namedEntitiesV3/*/category"
        },
        {
            "name": "confidence",
            "source":
                "/document/namedEntitiesV3/*/confidenceScore"
        }
    ]
}
],
"outputs": [
{
    "name": "output",
    "targetName": "namedEntities"
}
]
}

```

4. *(Optional)* If you're making use of the `entities` output to link entities to well-known entities, this feature is now a new skill, the [Microsoft.Skills.Text.V3.EntityLinkingSkill](#). Add the entity linking skill to your skillset to generate the linked entities. There are also a few minor changes to the property names of the `entities` output between the `EntityRecognitionSkill` and the new `EntityLinkingSkill`.

- a. `wikipediaId` is renamed to `id`
- b. `wikipediaLanguage` is renamed to `language`
- c. `wikipediaUrl` is renamed to `url`
- d. The `type` and `subtype` properties are no longer returned.

If you need to generate the exact same property names, add a [ShaperSkill](#) to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "LinkedEntitiesShaper",
    "description": "LinkedEntitiesShaper",
    "context": "/document/linkedEntitiesV3",
    "inputs": [

```

```

    {
        "name": "old_format",
        "sourceContext": "/document/linkedEntitiesV3/*",
        "inputs": [
            {
                "name": "name",
                "source": "/document/linkedEntitiesV3/*/name"
            },
            {
                "name": "wikipediaId",
                "source": "/document/linkedEntitiesV3/*/id"
            },
            {
                "name": "wikipediaLanguage",
                "source": "/document/linkedEntitiesV3/*/language"
            },
            {
                "name": "wikipediaUrl",
                "source": "/document/linkedEntitiesV3/*/url"
            },
            {
                "name": "bingId",
                "source": "/document/linkedEntitiesV3/*/bingId"
            },
            {
                "name": "matches",
                "source": "/document/linkedEntitiesV3/*/matches"
            }
        ]
    },
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}

```

5. (Optional) If you don't explicitly specify the `categories`, the

`EntityRecognitionSkill V3` can return different type of categories besides those that were supported by the `EntityRecognitionSkill`. If this behavior is undesirable, make sure to explicitly set the `categories` parameter to `["Person", "Location", "Organization", "Quantity", "Datetime", "URL", "Email"]`.

Sample Migration Definitions

- Simple migration

(Before) EntityRecognition skill definition

JSON

```
{  
    "@odata.type":  
        "#Microsoft.Skills.Text.EntityRecognitionSkill",  
    "categories": [ "Person" ],  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        }  
    ]  
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{  
    "@odata.type":  
        "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",  
    "categories": [ "Person" ],  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        }  
    ]  
}
```

- Complicated migration

(Before) EntityRecognition skill definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
  "categories": [ "Person", "Location", "Organization" ],
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.1,
  "includeTypelessEntities": true,
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    },
    {
      "name": "namedEntities",
      "targetName": "namedEntities"
    },
    {
      "name": "entities",
      "targetName": "entities"
    }
  ]
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "categories": [ "Person", "Location", "Organization" ],
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.1,
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    },
    {
      "name": "namedEntities",
      "targetName": "namedEntities"
    }
  ]
}
```

```
        "targetName": "namedEntitiesV3"
    }
]
},
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "NamedEntitiesShaper",
    "description": "NamedEntitiesShaper",
    "context": "/document/namedEntitiesV3",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/namedEntitiesV3/*",
            "inputs": [
                {
                    "name": "value",
                    "source": "/document/namedEntitiesV3/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntitiesV3/*/offset"
                },
                {
                    "name": "category",
                    "source":
                        "/document/namedEntitiesV3/*/category"
                },
                {
                    "name": "confidence",
                    "source":
                        "/document/namedEntitiesV3/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "namedEntities"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityLinkingSkill",
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "entities",
            "targetName": "entityLinks"
        }
    ]
}
```

```
        "targetName": "linkedEntities"
    }
]
},
{
"@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
"name": "LinkedEntitiesShaper",
"description": "LinkedEntitiesShaper",
"context": "/document/linkedEntitiesV3",
"inputs": [
{
    "name": "old_format",
    "sourceContext": "/document/linkedEntitiesV3/*",
    "inputs": [
        {
            "name": "name",
            "source": "/document/linkedEntitiesV3/*/name"
        },
        {
            "name": "wikipediaId",
            "source": "/document/linkedEntitiesV3/*/id"
        },
        {
            "name": "wikipediaLanguage",
            "source":
                "/document/linkedEntitiesV3/*/language"
        },
        {
            "name": "wikipediaUrl",
            "source": "/document/linkedEntitiesV3/*/url"
        },
        {
            "name": "bingId",
            "source":
                "/document/linkedEntitiesV3/*/bingId"
        },
        {
            "name": "matches",
            "source":
                "/document/linkedEntitiesV3/*/matches"
        }
    ]
},
{
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}
```

Microsoft.Skills.Text.SentimentSkill

Last available api version

2021-04-30-Preview

End of support

August 31, 2024

Recommendations

Use [Microsoft.Skills.Text.V3.SentimentSkill](#) instead. It provides an improved model and includes the option to add opinion mining or aspect-based sentiment. As the skill is significantly more complex, the outputs are also very different.

To migrate to the [Microsoft.Skills.Text.V3.SentimentSkill](#), make one or more of the following changes to your skill definition. You can update the skill definition using the [Update Skillset API](#).

Note

The skill outputs for the Sentiment Skill V3 are not compatible with the index definition based on the SentimentSkill. You will have to make changes to the index definition, skillset (later skill inputs and/or knowledge store projections) and indexer output field mappings to replace the sentiment skill with the new version.

1. *(Required)* Change the `@odata.type` from
"`#Microsoft.Skills.Text.SentimentSkill`" to
"`#Microsoft.Skills.Text.V3.SentimentSkill`".

2. *(Required)* The Sentiment Skill V3 provides a `positive`, `neutral`, and `negative` score for the overall text and the same scores for each sentence in the overall text, whereas the previous SentimentSkill only provided a single double that ranged from 0.0 (negative) to 1.0 (positive) for the overall text. You will need to update your index definition to accept the three double values in place of a single score, and make sure all of your downstream skill inputs, knowledge store projections, and output field mappings are consistent with the naming changes.

It's recommended to replace the old SentimentSkill with the SentimentSkill V3 entirely, update your downstream skill inputs, knowledge store projections, indexer output field mappings, and index definition to match the new output format, and reset your indexer so that all of your documents have consistent sentiment results going forward.

ⓘ Note

If you need any additional help updating your enrichment pipeline to use the latest version of the sentiment skill or if resetting your indexer is not an option for you, please open a [new support request](#) where we can work with you directly.

Microsoft.Skills.Text.NamedEntityRecognitionSkill

Last available api version

2017-11-11-Preview

End of support

August 31, 2024

Recommendations

Use [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#) instead. It provides most of the functionality of the NamedEntityRecognitionSkill at a higher quality. It also has richer information in its complex output fields.

To migrate to the [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#), make one or more of the following changes to your skill definition. You can update the skill definition using the [Update Skillset API](#).

1. *(Required)* Change the `@odata.type` from

```
"#Microsoft.Skills.Text.NamedEntityRecognitionSkill" to  
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill".
```

2. *(Optional)* If you're making use of the `entities` output, use the `namedEntities` complex collection output from the `EntityRecognitionSkill v3` instead. There are a few minor changes to the property names of the new `namedEntities` complex output:
 - a. `value` is renamed to `text`

- b. `confidence` is renamed to `confidenceScore`

If you need to generate the exact same property names, add a [ShaperSkill](#) to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",  
    "name": "NamedEntitiesShaper",  
    "description": "NamedEntitiesShaper",  
    "context": "/document/namedEntities",  
    "inputs": [  
        {  
            "name": "old_format",  
            "sourceContext": "/document/namedEntities/*",  
            "inputs": [  
                {  
                    "name": "value",  
                    "source": "/document/namedEntities/*/text"  
                },  
                {  
                    "name": "offset",  
                    "source": "/document/namedEntities/*/offset"  
                },  
                {  
                    "name": "category",  
                    "source": "/document/namedEntities/*/category"  
                },  
                {  
                    "name": "confidence",  
                    "source":  
                        "/document/namedEntities/*/confidenceScore"  
                }  
            ]  
        }  
    ],  
    "outputs": [  
        {  
            "name": "output",  
            "targetName": "entities"  
        }  
    ]  
}
```

3. (Optional) If you don't explicitly specify the `categories`, the `EntityRecognitionSkill v3` can return different type of categories besides those that were supported by the `NamedEntityRecognitionSkill`. If this behavior is

undesirable, make sure to explicitly set the `categories` parameter to `["Person", "Location", "Organization"]`.

Sample Migration Definitions

- Simple migration

(Before) NamedEntityRecognition skill definition

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",  
    "categories": [ "Person" ],  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        }  
    ]  
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",  
    "categories": [ "Person" ],  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        }  
    ]  
}
```

```
    ]  
}
```

- Slightly complicated migration

(Before) NamedEntityRecognition skill definition

JSON

```
{  
  "@odata.type":  
  "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",  
  "defaultLanguageCode": "en",  
  "minimumPrecision": 0.1,  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content"  
    }  
  ],  
  "outputs": [  
    {  
      "name": "persons",  
      "targetName": "people"  
    },  
    {  
      "name": "entities"  
    }  
  ]  
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{  
  "@odata.type":  
  "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",  
  "categories": [ "Person", "Location", "Organization" ],  
  "defaultLanguageCode": "en",  
  "minimumPrecision": 0.1,  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content"  
    }  
  ],  
  "outputs": [  
    {  
      "name": "persons",  
      "targetName": "people"  
    }  
  ]  
}
```

```

        },
        {
            "name": "namedEntities"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "NamedEntitiesShaper",
    "description": "NamedEntitiesShaper",
    "context": "/document/namedEntities",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/namedEntities/*",
            "inputs": [
                {
                    "name": "value",
                    "source": "/document/namedEntities/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntities/*/offset"
                },
                {
                    "name": "category",
                    "source": "/document/namedEntities/*/category"
                },
                {
                    "name": "confidence",
                    "source":
                        "/document/namedEntities/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}

```

See also

- Built-in skills
- How to define a skillset
- Entity Recognition Skill (V3)
- Sentiment Skill (V3)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Named Entity Recognition cognitive skill (v2)

The Named Entity Recognition skill (v2) extracts named entities from text. Available entities include the types `person`, `location` and `organization`.

Important

Named entity recognition skill (v2) (`Microsoft.Skills.Text.NamedEntityRecognitionSkill`) is now discontinued replaced by [Microsoft.Skills.Text.V3.EntityRecognitionSkill](#). Follow the recommendations in [Deprecated Azure AI Search skills](#) to migrate to a supported skill.

Note

As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to [attach a billable Microsoft Foundry resource](#). Charges accrue when calling APIs in Foundry Tools, and for image extraction as part of the document-cracking stage in Azure AI Search. There are no charges for text extraction from documents. Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#).

Image extraction is an extra charge metered by Azure AI Search, as described on the [pricing page](#). Text extraction is free.

@odata.type

`Microsoft.Skills.Text.NamedEntityRecognitionSkill`

Data limits

The maximum size of a record should be 50,000 characters as measured by [String.Length](#). If you need to break up your data before sending it to the key phrase extractor, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case sensitive.

[Expand table](#)

Parameter name	Description
categories	Array of categories that should be extracted. Possible category types: "Person", "Location", "Organization". If no category is provided, all types are returned.
defaultLanguageCode	Language code of the input text. The following languages are supported: de, en, es, fr, it
minimumPrecision	A number between 0 and 1. If the precision is lower than this value, the entity is not returned. The default is 0.

Skill inputs

[Expand table](#)

Input name	Description
languageCode	Optional. Default is "en".
text	The text to analyze.

Skill outputs

[Expand table](#)

Output name	Description
persons	An array of strings where each string represents the name of a person.
locations	An array of strings where each string represents a location.
organizations	An array of strings where each string represents an organization.
entities	An array of complex types. Each complex type includes the following fields: <ul style="list-style-type: none">• category ("person", "organization", or "location")• value (the actual entity name)• offset (The location where it was found in the text)• confidence (A value between 0 and 1 that represents that confidence that the value is an actual entity)

Sample definition

JSON

```
{  
    "@odata.type": "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",  
    "categories": [ "Person", "Location", "Organization"],  
    "defaultLanguageCode": "en",  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        }  
    ]  
}
```

Sample input

JSON

```
{  
    "values": [  
        {  
            "recordId": "1",  
            "data":  
                {  
                    "text": "This is the loan application for Joe Romero, a Microsoft  
employee who was born in Chile and who then moved to Australia.. Ana Smith is  
provided as a reference.",  
                    "languageCode": "en"  
                }  
        }  
    ]  
}
```

Sample output

JSON

```
{  
    "values": [  
        {  
            "recordId": "1",  
            "data":  
                {  
                    "text": "This is the loan application for Joe Romero, a Microsoft  
employee who was born in Chile and who then moved to Australia.. Ana Smith is  
provided as a reference.",  
                    "languageCode": "en"  
                }  
        }  
    ]  
}
```

```

"persons": [ "Joe Romero", "Ana Smith"],  

"locations": [ "Chile", "Australia"],  

"organizations": [ "Microsoft"],  

"entities":  

[  

  {  

    "category": "person",  

    "value": "Joe Romero",  

    "offset": 33,  

    "confidence": 0.87  

  },  

  {  

    "category": "person",  

    "value": "Ana Smith",  

    "offset": 124,  

    "confidence": 0.87  

  },  

  {  

    "category": "location",  

    "value": "Chile",  

    "offset": 88,  

    "confidence": 0.99  

  },  

  {  

    "category": "location",  

    "value": "Australia",  

    "offset": 112,  

    "confidence": 0.99  

  },  

  {  

    "category": "organization",  

    "value": "Microsoft",  

    "offset": 54,  

    "confidence": 0.99  

  }
]
}
]
}

```

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

- Entity Recognition Skill (V3)
-

Last updated on 11/18/2025

Entity Recognition cognitive skill (v2)

The Entity Recognition skill (v2) extracts entities of different types from text. This skill uses the machine learning models provided by [Text Analytics](#) in Foundry Tools.

Important

The Entity Recognition skill (v2) (`Microsoft.Skills.Text.EntityRecognitionSkill`) is now discontinued replaced by [`Microsoft.Skills.Text.V3.EntityRecognitionSkill`](#). Follow the recommendations in [Deprecated skills](#) to migrate to a supported skill.

Note

As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to [attach a billable Microsoft Foundry resource](#). Charges accrue when calling APIs in Foundry Tools, and for image extraction as part of the document-cracking stage in Azure AI Search. There are no charges for text extraction from documents.

Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#). Image extraction pricing is described on the [Azure AI Search pricing page](#).

@odata.type

`Microsoft.Skills.Text.EntityRecognitionSkill`

Data limits

The maximum size of a record should be 50,000 characters as measured by [`String.Length`](#). If you need to break up your data before sending it to the key phrase extractor, consider using the [Text Split skill](#). If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case sensitive and are all optional.

 Expand table

Parameter name	Description
categories	Array of categories that should be extracted. Possible category types: "Person", "Location", "Organization", "Quantity", "Datetime", "URL", "Email". If no category is provided, all types are returned.
defaultLanguageCode	Language code of the input text. The following languages are supported: ar, cs, da, de, en, es, fi, fr, hu, it, ja, ko, nl, no, pl, pt-BR, pt-PT, ru, sv, tr, zh-hans. Not all entity categories are supported for all languages; see note below.
minimumPrecision	A value between 0 and 1. If the confidence score (in the <code>namedEntities</code> output) is lower than this value, the entity is not returned. The default is 0.
includeTypelessEntities	Set to <code>true</code> if you want to recognize well-known entities that don't fit the current categories. Recognized entities are returned in the <code>entities</code> complex output field. For example, "Windows 10" is a well-known entity (a product), but since "Products" is not a supported category, this entity would be included in the <code>entities</code> output field. Default is <code>false</code> .

Skill inputs

[+] Expand table

Input name	Description
languageCode	Optional. Default is "en".
text	The text to analyze.

Skill outputs

! Note

Not all entity categories are supported for all languages. The "Person", "Location", and "Organization" entity category types are supported for the full list of languages above. Only *de*, *en*, *es*, *fr*, and *zh-hans* support extraction of "Quantity", "Datetime", "URL", and "Email" types. For more information, see [Language and region support for the Text Analytics API](#).

[+] Expand table

Output name	Description
persons	An array of strings where each string represents the name of a person.
locations	An array of strings where each string represents a location.
organizations	An array of strings where each string represents an organization.
quantities	An array of strings where each string represents a quantity.
dateTimes	An array of strings where each string represents a DateTime (as it appears in the text) value.
urls	An array of strings where each string represents a URL
emails	An array of strings where each string represents an email
namedEntities	An array of complex types that contains the following fields: <ul style="list-style-type: none"> category value (The actual entity name) offset (The location where it was found in the text) confidence (Higher value means it's more to be a real entity)
entities	An array of complex types that contains rich information about the entities extracted from text, with the following fields <ul style="list-style-type: none"> name (the actual entity name. This represents a "normalized" form) wikipediaId wikipediaLanguage wikipediaUrl (a link to Wikipedia page for the entity) bingId type (the category of the entity recognized) subType (available only for certain categories, this gives a more granular view of the entity type) matches (a complex collection that contains) <ul style="list-style-type: none"> text (the raw text for the entity) offset (the location where it was found) length (the length of the raw entity text)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
  "categories": [ "Person", "Email" ],
  "defaultLanguageCode": "en",
  "includeTypelessEntities": true,
  "minimumPrecision": 0.5,
```

```
"inputs": [
  {
    "name": "text",
    "source": "/document/content"
  }
],
"outputs": [
  {
    "name": "persons",
    "targetName": "people"
  },
  {
    "name": "emails",
    "targetName": "contact"
  },
  {
    "name": "entities"
  }
]
```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "Contoso corporation was founded by John Smith. They can be reached at contact@contoso.com",
        "languageCode": "en"
      }
    }
  ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "
```

```
"persons": [ "John Smith"],  
"emails": ["contact@contoso.com"],  
"namedEntities":  
[  
  {  
    "category": "Person",  
    "value": "John Smith",  
    "offset": 35,  
    "confidence": 0.98  
  }  
],  
"entities":  
[  
  {  
    "name": "John Smith",  
    "wikipediaId": null,  
    "wikipediaLanguage": null,  
    "wikipediaUrl": null,  
    "bingId": null,  
    "type": "Person",  
    "subType": null,  
    "matches": [  
      {  
        "text": "John Smith",  
        "offset": 35,  
        "length": 10  
      }]  
    },  
    {  
      "name": "contact@contoso.com",  
      "wikipediaId": null,  
      "wikipediaLanguage": null,  
      "wikipediaUrl": null,  
      "bingId": null,  
      "type": "Email",  
      "subType": null,  
      "matches": [  
        {  
          "text": "contact@contoso.com",  
          "offset": 70,  
          "length": 19  
        }]  
      ],  
      {  
        "name": "Contoso",  
        "wikipediaId": "Contoso",  
        "wikipediaLanguage": "en",  
        "wikipediaUrl": "https://en.wikipedia.org/wiki/Contoso",  
        "bingId": "349f014e-7a37-e619-0374-787ebb288113",  
        "type": null,  
        "subType": null,  
        "matches": [  
          {  
            "text": "Contoso",  
            "offset": 0,  
            "length": 7
```

```
        }]
    ]
}
]
}
```

Note that the offsets returned for entities in the output of this skill are directly returned from the [Text Analytics API](#), which means if you are using them to index into the original string, you should use the [StringInfo](#) class in .NET in order to extract the correct content. [More details can be found here.](#)

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Entity Recognition Skill \(V3\)](#)

Last updated on 11/18/2025

Sentiment cognitive skill (v2)

The Sentiment skill (v2) evaluates unstructured text along a positive-negative continuum, and for each record, returns a numeric score between 0 and 1. Scores close to 1 indicate positive sentiment, and scores close to 0 indicate negative sentiment. This skill uses the machine learning models provided by [Text Analytics](#) in Foundry Tools.

Important

The Sentiment skill (v2) (`Microsoft.Skills.Text.SentimentSkill`) is now discontinued replaced by [`Microsoft.Skills.Text.V3.SentimentSkill`](#). Follow the recommendations in [Deprecated Azure AI Search skills](#) to migrate to a supported skill.

Note

As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to [attach a billable Microsoft Foundry resource](#). Charges accrue when calling APIs in Foundry Tools, and for image extraction as part of the document-cracking stage in Azure AI Search. There are no charges for text extraction from documents.

Execution of built-in skills is charged at the existing [Foundry Tools Standard price](#). Image extraction pricing is described on the [Azure AI Search pricing page](#).

@odata.type

`Microsoft.Skills.Text.SentimentSkill`

Data limits

The maximum size of a record should be 5000 characters as measured by [`String.Length`](#). If you need to break up your data before sending it to the sentiment analyzer, use the [`Text Split` skill](#).

Skill parameters

Parameters are case sensitive.

 Expand table

Parameter Name	Description
defaultLanguageCode	(optional) The language code to apply to documents that don't specify language explicitly. See the full list of supported languages .

Skill inputs

[] [Expand table](#)

Input Name	Description
text	The text to be analyzed.
languageCode	(Optional) A string indicating the language of the records. If this parameter is not specified, the default value is "en". See the full list of supported languages .

Skill outputs

[] [Expand table](#)

Output Name	Description
score	A value between 0 and 1 that represents the sentiment of the analyzed text. Values close to 0 have negative sentiment, close to 0.5 have neutral sentiment, and values close to 1 have positive sentiment.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    },
    {
      "name": "languageCode",
      "source": "/document/languagecode"
    }
  ],
}
```

```
"outputs": [
  {
    "name": "score",
    "targetName": "mySentiment"
  }
]
```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "I had a terrible time at the hotel. The staff was rude and
the food was awful.",
        "languageCode": "en"
      }
    }
  ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "score": 0.01
      }
    }
  ]
}
```

Warning cases

If your text is empty, a warning is generated and no sentiment score is returned. If a language is not supported, a warning is generated and no sentiment score is returned.

See also

- [Built-in skills](#)
 - [How to define a skillset](#)
 - [Sentiment Skill \(V3\)](#)
-

Last updated on 11/18/2025

Microsoft Foundry model catalog vectorizer

ⓘ Important

This vectorizer is in public preview under [Supplemental Terms of Use](#). To use this feature, we recommend the latest preview version of [Indexes - Create Or Update \(REST API\)](#).

The Microsoft Foundry model catalog vectorizer connects to an embedding model deployed from the [Foundry model catalog](#) or an [Azure Machine Learning](#) (AML) endpoint. Your data is processed in the [Geo](#) where your model is deployed.

If you're using integrated vectorization to create the vector arrays, the skillset should include an [AML skill](#) that points to the same model specified in the vectorizer.

Prerequisites

- A [Foundry hub-based project](#) or an [AML workspace](#) for a custom model that you create.
- For hub-based projects only, a [serverless deployment](#) of a [supported model](#) from the Foundry model catalog.

Vectorizer parameters

Parameters are case sensitive. The parameters you use depend on what [authentication](#) your model provider requires, if any.

[+] [Expand table](#)

Parameter	Description
name	
<code>uri</code>	(Required for key authentication) The target URI of the serverless deployment from the Foundry model catalog or the scoring URI of the AML online endpoint . Only the HTTPS URI scheme is allowed.
<code>key</code>	(Required for key authentication) The API key of the model provider.
<code>resourceId</code>	(Required for token authentication) The Azure Resource Manager resource ID of the model provider. For an AML online endpoint, use the <code>subscriptions/{guid}/resourceGroups/{resource-group-}</code>

Parameter	Description
<code>name</code>	
	<code>name}/Microsoft.MachineLearningServices/workspaces/{workspace-name}/onlineendpoints/{endpoint_name}</code> format.
<code>modelName</code>	The name of the embedding model from the Foundry model catalog deployed at the specified <code>uri</code> . Supported models are: <ul style="list-style-type: none"> • Cohere-embed-v3-english • Cohere-embed-v3-multilingual • Cohere-embed-v4
<code>region</code>	(Optional for token authentication) The region in which the model provider is deployed. Required if the region is different from the region of the search service.
<code>timeout</code>	(Optional) The timeout for the HTTP client making the API call. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). For example, <code>PT60S</code> for 60 seconds. If not set, a default value of 30 seconds is chosen. The timeout can be set to a maximum of 230 seconds and a minimum of 1 second.

What authentication parameters to use

The Microsoft Foundry model catalog vectorizer provides two authentication options:

- **Key-based authentication.** You provide a static key to authenticate scoring requests from the vectorizer. Set the `uri` and `key` parameters for this connection.
- **Token-based authentication.** The Foundry hub-based project or AML online endpoint is deployed using token-based authentication. The Azure AI Search service must have a [managed identity](#) and a role assignment on the model provider. The vectorizer then uses the search service identity to authenticate against the model provider, with no static keys required. The search service identity must have the **Owner** or **Contributor** role. Set the `resourceId` parameter, and if the search service is in a different region from the model provider, set the `region` parameter.

Supported vector query types

Which vector query types are supported by the Microsoft Foundry model catalog vectorizer depends on the `modelName` that is configured.

[] [Expand table](#)

Embedding model	Supports <code>text</code> query	Supports <code>imageUrl</code> query	Supports <code>imageBinary</code> query
Cohere-embed-v3-english	X		X
Cohere-embed-v3-multilingual	X		X
Cohere-embed-v4	X		X

Expected field dimensions

The expected field dimensions for a vector field configured with a Microsoft Foundry model catalog vectorizer depend on the `modelName` that is configured.

[Expand table](#)

<code>modelName</code>	Expected dimensions
Cohere-embed-v3-english	1024
Cohere-embed-v3-multilingual	1024
Cohere-embed-v4	256–1536

Sample definition

Suggested model names in the Foundry model catalog consist of the base model plus a random three-letter suffix. The name of your model will be different from the one shown in this example.

JSON

```
"vectorizers": [
  {
    "name": "my-model-catalog-vectorizer",
    "kind": "aml",
    "amlParameters": {
      "uri": "https://Cohere-embed-v3-multilingual-
hin.eastus.models.ai.azure.com",
      "key": "aaaaaaaa-0b0b-1c1c-2d2d-333333333333",
      "timeout": "PT60S",
      "modelName": "Cohere-embed-v3-multilingual-hin",
      "resourceId": null,
      "region": null,
    },
  },
]
```

```
    }  
]
```

See also

- [Integrated vectorization](#)
- [Integrated vectorization with models from Foundry](#)
- [Configure a vectorizer in a search index](#)
- [AML skill](#)
- [Foundry model catalog](#)

Last updated on 11/18/2025

Azure OpenAI vectorizer

The Azure OpenAI vectorizer connects to an embedding model deployed to your [Azure OpenAI in Foundry Models](#) resource or [Microsoft Foundry](#) project to generate embeddings at query time. Your data is processed in the [Geo ↗](#) where your model is deployed.

Although vectorizers are used at query time, you specify them in index definitions and reference them on vector fields through a vector profile. For more information, see [Configure a vectorizer in a search index](#).

The Azure OpenAI vectorizer is called `AzureOpenAIVectorizer` in the REST API. Use the latest stable version of [Indexes - Create \(REST API\)](#) or an Azure SDK package that provides the feature.

(!) Note

This vectorizer is bound to Azure OpenAI and is charged at the [Azure OpenAI Standard price ↗](#).

Prerequisites

- An [Azure OpenAI in Foundry Models resource](#) or Foundry project.
 - Your Azure OpenAI resource must have a [custom subdomain](#), such as `https://<resource-name>.openai.azure.com`. You can find this endpoint on the **Keys and Endpoint** page in the Azure portal and use it for the `resourceUri` property in this skill.
 - The [parent resource](#) of your Foundry project provides access to multiple endpoints, including `https://<resource-name>.openai.azure.com`, `https://<resource-name>.services.ai.azure.com`, and `https://<resource-name>.cognitiveservices.azure.com`. You can find these endpoints on the **Keys and Endpoint** page in the Azure portal and use any of them for the `resourceUri` property in this skill.
- An Azure OpenAI embedding model deployed to your resource or project. For supported models, see the next section.

Vectorizer parameters

Parameters are case sensitive.

Parameter	Description
name	
<code>resourceUri</code>	(Required) The URI of the model provider. Supported domains are: <ul style="list-style-type: none"> • <code>openai.azure.com</code> • <code>services.ai.azure.com</code> • <code>cognitiveservices.azure.com</code> <p>Azure API Management endpoints are supported with URL <code>https://<resource-name>.azure-api.net</code>. Shared private links aren't supported for API Management endpoints.</p>
<code>apiKey</code>	The secret key used to access the model. If you provide a key, leave <code>authIdentity</code> empty. If you set both <code>apiKey</code> and <code>authIdentity</code> , the <code>apiKey</code> is used on the connection.
<code>deploymentId</code>	(Required) The ID of the deployed Azure OpenAI embedding model. This is the deployment name you specified when you deployed the model.
<code>authIdentity</code>	A user-managed identity used by the search service for the connection. You can use either a system- or user-managed identity . To use a system-managed identity, leave <code>apiKey</code> and <code>authIdentity</code> blank. The system-managed identity is used automatically. A managed identity must have Cognitive Services OpenAI User permissions to send text to Azure OpenAI.
<code>modelName</code>	(Required) The name of the Azure OpenAI model deployed at the specified <code>deploymentId</code> . Supported values are: <ul style="list-style-type: none"> • <code>text-embedding-ada-002</code> • <code>text-embedding-3-large</code> • <code>text-embedding-3-small</code>

Supported vector query types

The Azure OpenAI vectorizer only supports `text` vector queries.

Expected field dimensions

The expected field dimensions for a field configured with an Azure OpenAI vectorizer depend on the `modelName` that is configured.

modelName	Minimum dimensions	Maximum dimensions
text-embedding-ada-002	1536	1536
text-embedding-3-large	1	3072
text-embedding-3-small	1	1536

Sample definition

JSON

See also

- Integrated vectorization
 - How to configure a vectorizer in a search index
 - Azure OpenAI Embedding skill

Last updated on 11/18/2025

Azure Vision vectorizer

Important

This vectorizer is in public preview under [Supplemental Terms of Use](#). The [2024-05-01-Preview REST API](#) and newer preview APIs support this feature.

The **Azure Vision** vectorizer connects to Azure Vision in Foundry Tools via a [Microsoft Foundry resource](#). At query time, the vectorizer uses the [multimodal embeddings API](#) to generate embeddings.

To determine where this model is accessible, see the [region availability for multimodal embeddings](#). Your data is processed in the [Geo](#) where your model is deployed.

Note

This vectorizer is bound to Foundry Tools. Execution of the vectorizer is charged at the [Foundry Tools Standard price](#).

Vectorizer parameters

Parameters are case sensitive.

 [Expand table](#)

Parameter	Description
<code>name</code>	
<code>resourceUri</code>	The endpoint of the Foundry resource, which must have the the <code>https://<resource-name>.services.ai.azure.com</code> or <code>https://<resource-name>.cognitiveservices.azure.com</code> format. You can find this endpoint on the Keys and Endpoint page in the Azure portal.
<code>apiKey</code>	The API key of the Foundry resource.
<code>modelVersion</code>	(Required) The model version to be passed to the Azure Vision API for generating embeddings. It's important that all embeddings stored in a given index field are generated using the same <code>modelVersion</code> . For information about version support for this model refer to multimodal embeddings .
<code>authIdentity</code>	A user-managed identity used by the search service for connecting to Foundry. You can use either a system- or user-managed identity . To use a system-managed identity, leave <code>apiKey</code> and <code>authIdentity</code> blank. The system-managed identity is used automatically. A managed identity must have Cognitive Services User permissions to use this vectorizer.

Supported vector query types

The Azure Vision vectorizer supports `text`, `imageUrl`, and `imageBinary` vector queries.

Expected field dimensions

A vector field configured with the Azure Vision vectorizer should have a `dimensions` value of 1024.

Sample definition

JSON

```
"vectorizers": [
  {
    "name": "my-ai-services-vision-vectorizer",
    "kind": "aiServicesVision",
    "aiServicesVisionParameters": {
      "resourceUri": "https://westus.api.cognitive.microsoft.com/",
      "apiKey": "00000000000000000000000000000000",
      "authIdentity": null,
      "modelVersion": "2023-04-15"
    }
  }
]
```

See also

- [Integrated vectorization](#)
- [How to configure a vectorizer in a search index](#)
- [Azure Vision multimodal embeddings skill](#)

Last updated on 11/18/2025

Custom Web API vectorizer

09/12/2025

The **custom web API vectorizer** allows you to configure your search queries to call out to a Web API endpoint to generate embeddings at query time. The structure of the JSON payload required to be implemented in the provided endpoint is described further down in this document. Your data is processed in the [Geo](#) where your model is deployed.

Although vectorizers are used at query time, you specify them in index definitions and reference them on vector fields through a vector profile. For more information, see [Configure a vectorizer in a search index](#).

The custom web API vectorizer is called `WebApiVectorizer` in the REST API. Use the latest stable version of [Indexes - Create \(REST API\)](#) or an Azure SDK package that provides the feature.

Vectorizer parameters

Parameters are case-sensitive.

 Expand table

Parameter	Description
<code>name</code>	
<code>uri</code>	The URI of the Web API to which the JSON payload is sent. Only the <code>https</code> URI scheme is allowed.
<code>httpMethod</code>	The method to use while sending the payload. Allowed methods are <code>PUT</code> or <code>POST</code>
<code>httpHeaders</code>	A collection of key-value pairs where the keys represent header names and values represent header values that are sent to your Web API along with the payload. The following headers are prohibited from being in this collection: <code>Accept</code> , <code>Accept-Charset</code> , <code>Accept-Encoding</code> , <code>Content-Length</code> , <code>Content-Type</code> , <code>Cookie</code> , <code>Host</code> , <code>TE</code> , <code>Upgrade</code> , <code>Via</code> .
<code>authResourceId</code>	(Optional) A string that if set, indicates that this vectorizer should use a managed identity on the connection to the function or app hosting the code. This property takes an application (client) ID or app's registration in Microsoft Entra ID, in any of these formats: <code>api://<appId></code> , <code><appId>/.default</code> , <code>api://<appId>/.default</code> . This value is used to scope the authentication token retrieved by the indexer, and is sent along with the custom Web API request to the function or app. Setting this property requires that your search service is configured for managed identity and your Azure function app is configured for a Microsoft Entra sign in .
<code>authIdentity</code>	(Optional) A user-managed identity used by the search service for connecting to the function or app hosting the code. You can use either a system or user managed

Parameter	Description
<code>name</code>	
<code>timeout</code>	(Optional) When specified, indicates the timeout for the http client making the API call. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). For example, <code>PT60S</code> for 60 seconds. If not set, a default value of 30 seconds is chosen. The timeout can be set to a maximum of 230 seconds and a minimum of 1 second.

Supported vector query types

The Custom Web API vectorizer supports `text`, `imageUrl`, and `imageBinary` vector queries.

Sample definition

JSON

```
"vectorizers": [
  {
    "name": "my-custom-web-api-vectorizer",
    "kind": "customWebApi",
    "customWebApiParameters": {
      "uri": "https://contoso.embeddings.com",
      "httpMethod": "POST",
      "httpHeaders": {
        "api-key": "00000000000000000000000000000000"
      },
      "timeout": "PT60S",
      "authResourceId": null,
      "authIdentity": null
    }
  }
]
```

JSON payload structure

The required JSON payload structure that is expected for an endpoint when using it with the custom web API vectorizer is the same as that of the custom web API skill, which is discussed in more detail in [the documentation for the skill](#).

There are the following other considerations to make when implementing a web API endpoint to use with the custom web API vectorizer.

- The vectorizer sends only one record at a time in the `values` array when making a request to the endpoint.
- The vectorizer passes the data to be vectorized in a specific key in the `data` JSON object in the request payload. That key is `text`, `imageUrl`, or `imageBinary`, depending on which type of vector query was requested.
- The vectorizer expects the resulting embedding to be under the `vector` key in the `data` JSON object in the response payload.
- Any errors or warnings returned by the endpoint are ignored by the vectorizer and not obtainable for debugging purposes at query time.
- If an `imageBinary` vector query was requested, the request payload sent to the endpoint is the following:

```
JSON

{
  "values": [
    {
      "recordId": "0",
      "data": {
        "imageBinary": {
          "data": "<base 64 encoded image binary data>"
        }
      }
    }
  ]
}
```

See also

- [Integrated vectorization](#)
- [How to configure a vectorizer in a search index](#)
- [Custom Web API skill](#)
- [Hugging Face Embeddings Generator power skill \(can be used for a custom web API vectorizer as well\)](#) ↗

Azure compliance documentation

If your organization needs to comply with legal or regulatory standards, start here to learn about compliance in Azure.

Compliance offerings

Global

- [!\[\]\(b9d5a7cb9c33d7466c916da7a319c6a1_img.jpg\) CIS benchmark](#)
- [!\[\]\(f10a13190c588dc3d689b584974c3169_img.jpg\) CSA STAR Attestation](#)
- [!\[\]\(aabb372d52e6ad8648ebacba6c09a4b1_img.jpg\) CSA STAR Certification](#)
- [!\[\]\(241027e11f8e3dc95c768735f46352d5_img.jpg\) CSA STAR self-assessment](#)
- [!\[\]\(aedd8117f83641f30b48f26d4249531c_img.jpg\) SOC 1](#)
- [!\[\]\(82e242de304d3fff567b5b035838dacc_img.jpg\) SOC 2](#)
- [!\[\]\(2d1df36d56f20090e84e2657f0a7742d_img.jpg\) SOC 3](#)

Global

- [!\[\]\(816e47288d1678b5758a6b28ba005d8c_img.jpg\) ISO 20000-1](#)
- [!\[\]\(05097d36ad909479a0870faa822d0c26_img.jpg\) ISO 22301](#)
- [!\[\]\(78486de7dc191ceab27b513aad5ccdd6_img.jpg\) ISO 27001](#)
- [!\[\]\(ed73354a7d37b2b259b5a9e8c4ba4240_img.jpg\) ISO 27017](#)
- [!\[\]\(ea768df331f2195a91501d90e13d9737_img.jpg\) ISO 27018](#)
- [!\[\]\(9987be61f77d10e379d8491284acee04_img.jpg\) ISO 27701](#)
- [!\[\]\(ea5cc1e05f03d09e5e833c2b4ed07b32_img.jpg\) ISO 9001](#)
- [!\[\]\(cd9b6788743ee2c33d9914cceca3be89_img.jpg\) WCAG](#)

US government

- [!\[\]\(7461357c31cfc43ff49cc46bc1ad57f5_img.jpg\) CJIS](#)
- [!\[\]\(aaf46b6a94a7091d91a7717b9cb266b1_img.jpg\) CMMC](#)
- [!\[\]\(53e1c5db73f9590650432204003ac2c3_img.jpg\) CNSSI 1253](#)
- [!\[\]\(782d6768f4d7094175f2f93092d3bf8e_img.jpg\) DFARS](#)
- [!\[\]\(cc53c25ddf78be66e2f43f6cf8d05d19_img.jpg\) DoD IL2](#)
- [!\[\]\(6f94dd6429eb615a09c60bf5b487c169_img.jpg\) DoD IL4](#)
- [!\[\]\(96a4738fa7362c1cc38825df83576ae7_img.jpg\) DoD IL5](#)
- [!\[\]\(d705c34016d3a8a77666da082b17010e_img.jpg\) DoD IL6](#)
- [!\[\]\(40df32c7d14c71a087f74a3de0b13cfc_img.jpg\) DoE 10 CFR Part 810](#)
- [!\[\]\(1ad43f2c8d76ab50981799e787aede3d_img.jpg\) EAR](#)
- [!\[\]\(ddf4bb544874e4922f24cde4fcb09279_img.jpg\) FedRAMP](#)
- [!\[\]\(361234d4fe86d4115630831c368526c6_img.jpg\) FIPS 140](#)

US government

- [!\[\]\(1187415f0e9ec83c2fa3bd67208df422_img.jpg\) ICD 503](#)
- [!\[\]\(b433c3082930c20061e1982c9703bf2e_img.jpg\) IRS 1075](#)
- [!\[\]\(bda07746a59d4f6a3dc5a4022a40c243_img.jpg\) ITAR](#)
- [!\[\]\(1b1eaebcf3ae64883c4ba5d74d08f3d4_img.jpg\) JSIG](#)
- [!\[\]\(a60a10fffd96646f62b88976b05f37b0_img.jpg\) NDAA](#)
- [!\[\]\(7c184d1ce90f22a72ee1f2a33f69241d_img.jpg\) NIST 800-161](#)
- [!\[\]\(e26a1986b4325ab3c75a70f62b806af8_img.jpg\) NIST 800-171](#)
- [!\[\]\(e0591f0fa7c08ef13861cfb37282e899_img.jpg\) NIST 800-53](#)
- [!\[\]\(91af4ffe7f3d09be9b0ae35d301cc90c_img.jpg\) NIST 800-63](#)
- [!\[\]\(346a2e1594bce4aa56eab948f7ba214c_img.jpg\) NIST CSF](#)
- [!\[\]\(9e6291478d0501247109f695de551557_img.jpg\) Section 508 VPATs](#)
- [!\[\]\(9123542c602271b38131e2f2e0f186b8_img.jpg\) StateRAMP](#)

Financial services

 23 NYCRR Part 500 (US)

 AFM and DNB (Netherlands)

 AMF and ACPR (France)

 APRA (Australia)

 CFTC 1.31 (US)

 EBA (EU)

 FCA and PRA (UK)

 FFIEC (US)

 FINMA (Switzerland)

Financial services

 FINRA 4511 (US)

 FISC (Japan)

 FSA (Denmark)

 GLBA (US)

 KNF (Poland)

 MAS and ABS (Singapore)

 NBB and FSMA (Belgium)

 OSFI (Canada)

Financial services

 OSPAR (Singapore)

 PCI 3DS

 PCI DSS

 RBI and IRDAI (India)

 SEC 17a-4 (US)

 SEC Regulation SCI (US)

 SOX (US)

 TruSight

Healthcare and life sciences

 ASIP HDS (France)

 EPICS (US)

 GxP (FDA 21 CFR Part 11)

 HIPAA (US)

 HITRUST

 MARS-E (US)

 NEN 7510 (Netherlands)

Automotive, education, energy, media, and telecommunication

 CDSA

 DPP (UK)

 FACT (UK)

 FERPA (US)

 MPA

 GSMA

 NERC (US)

 TISAX

Regional - Americas

 Argentina PDPA

 Canada privacy laws

 Canada Protected B

 US CCPA

Regional - Asia Pacific

 Australia IRAP

 China GB 18030

Regional - EMEA

 EU Cloud CoC

 EU EN 301 549

- [China DJCP \(MLPS\)](#)
- [China TCS](#)
- [India MeitY](#)
- [Japan CS Gold Mark](#)
- [Japan ISMAP](#)
- [Japan My Number Act](#)
- [Korea K-ISMS](#)
- [Korea CSAP](#)
- [New Zealand ISPC](#)
- [Singapore MTCS](#)

- [ENISA IAF](#)
- [EU GDPR](#)
- [EU Model Clauses](#)
- [Germany C5](#)
- [Germany IT-Grundsatz workbook](#)
- [Netherlands BIR 2012](#)
- [Qatar NIA](#)

Regional - EMEA

- [Russia personal data law](#)
- [Spain ENS High](#)
- [UAE DESC](#)
- [UK Cyber Essentials Plus](#)
- [UK G-Cloud](#)
- [UK PASF](#)

More compliance resources

To access a resource, you may need to be signed into your cloud service.

Privacy and GDPR

- [Checklists](#)
- [Data subject requests](#)
- [Breach notification](#)
- [Data protection impact assessments](#)
- [Data residency in Azure ↗](#)

Azure Policy regulatory compliance built-in initiatives

- [Australian Government ISM PROTECTED](#)
- [Canada Federal PBMM](#)
- [CIS Azure Foundations Benchmark](#)
- [FedRAMP High](#)
- [HIPAA HITRUST](#)
- [IRS 1075](#)
- [ISO 27001](#)

Country/Region privacy and compliance guides

- [Australian security and privacy requirements ↗](#)
- [Singapore security and privacy requirements ↗](#)
- [Japan security and privacy requirements ↗](#)
- [New Zealand security and privacy requirements ↗](#)
- [Navigating your way to the cloud in Europe ↗](#)
- [Navigating to the cloud in Middle East and Africa ↗](#)

[PCI DSS](#)
[NIST SP 800-171](#)
[UK OFFICIAL and UK NHS](#)

[Cloud compliance guides for financial services ↗](#)

Implementation and mappings

[Azure Policy Regulatory Compliance \(preview\)](#)
[CIS Azure Foundations Benchmark ↗](#)
[CSA CAIQ ↗](#)
[FERPA implementation guide ↗](#)
[GDPR control mapping ↗](#)
[GxP guidelines ↗](#)
[HITRUST customer responsibility matrix ↗](#)

Implementation and mappings

[ISO 27001 security controls ↗](#)
[IT-Grundschutz workbook \(German\)](#)
[NERC CIP standards and cloud computing](#)
[NZ GCIO cloud computing considerations ↗](#)
[PCI 3DS attestation documents ↗](#)
[SEC Regulation SCI guidance ↗](#)
[SOX guidance ↗](#)
[UK OFFICIAL cloud security controls ↗](#)

White papers and analyst reports

[Overview of Azure compliance](#)
[Enabling data residency and protection ↗](#)
[Azure for worldwide public sector](#)
[Azure Internet of Things compliance ↗](#)
[IDC - Azure manages regulatory challenges ↗](#)
[Azure risk compliance guide ↗](#)
[Shared responsibilities for cloud computing ↗](#)
[Azure export controls ↗](#)
[Azure enables a world of compliance ↗](#)
[Azure cloud platform for PCI 3DS ↗](#)
[A practical guide to designing secure health solutions ↗](#)

Productivity tools and accelerators for Azure AI Search

07/28/2025

Microsoft engineers build productivity tools that aren't part of the Azure AI Search service and aren't covered by service-level agreements (SLAs). You can download, modify, and build these tools to create an app that helps you develop or maintain a search solution.

Accelerators

 Expand table

Accelerator	Description
RAG Experiment Accelerator	Conduct experiments and evaluations using Azure AI Search and the RAG pattern. This accelerator has code for loading multiple data sources, using a variety of models, and creating a variety of search indexes and queries.
Build your own copilot solution accelerator	Code and docs to build a copilot for specific use cases.
Chat with your data solution accelerator	Code and docs to create interactive search solution in production environments.
Document knowledge mining solution accelerator	Code and docs built on Azure OpenAI in Azure AI Foundry Models and Azure AI Document Intelligence. It processes and extracts summaries, entities, and metadata from unstructured, multimodal documents to enable searching and chatting over this data.

Tools

 Expand table

Tool name	Description
Azure AI Search Lab	Learning and experimentation lab to try out AI-enabled search scenarios in Azure. It provides a web application front end that uses Azure AI Search and Azure OpenAI to execute searches with various options. These options range from simple keyword search to semantic ranking, vector and hybrid search, and using generative AI to answer search queries.
Back up and restore (C#)	Download the retrievable fields of an index to your local device and then upload the index and its content to a new search service.

Tool name	Description
Back up and restore (Python)	Download the retrievable fields of an index to your local device and then upload the index and its content to a new search service.
Performance testing solution	This solution helps you load test Azure AI Search. It uses Apache JMeter as an open source load and performance testing tool and Terraform to dynamically provision and destroy the required infrastructure on Azure.
Visual Studio Code extension	Although the extension is no longer available on the Visual Studio Code Marketplace, the code is open source. You can clone and modify the tool for your own use.

Training - Azure AI Search

Training modules deliver an end-to-end experience that helps you build skills and develop insights while working through a progression of exercises. Visit the following links to begin learning with prepared lessons from Microsoft and other training providers.

Learning paths

Learning paths are a collection of training modules that are organized around specific roles (like developer) or technologies (like Azure AI Search). Azure AI Search is covered in two learning paths.

- [Microsoft Azure AI Fundamentals: Azure Document Intelligence in Foundry Tools and Knowledge Mining](#) includes two modules: [Fundamentals of Azure AI Search](#) and [knowledge mining](#) and [Fundamentals of Azure Document Intelligence](#). Choose this learning path to broaden your understanding of when to use each technology.
- [Implement knowledge mining with Azure AI Search](#) includes eight modules that teach important skills. Choose this learning path to gain expertise in Azure AI Search.

Modules

 Expand table

Module	Learning path
Fundamentals of Knowledge Mining and Azure AI Search	Microsoft Azure AI Fundamentals
Create an Azure AI Search solution	Implement knowledge mining
Create a custom skill for Azure AI Search	Implement knowledge mining
Enrich your data with Azure Language in Foundry Tools	
Create a knowledge store with Azure AI Search	Implement knowledge mining
Implement advanced search features in Azure AI Search	Implement knowledge mining
Search data outside the Azure platform in Azure AI Search using Azure Data Factory	Implement knowledge mining
Perform vector search and retrieval in Azure AI Search	Implement knowledge mining
Perform search reranking with semantic ranking in Azure AI Search	Implement knowledge mining

Module	Learning path
Maintain an Azure AI Search solution	Implement knowledge mining

RAG-centric modules

- [Build a RAG-based copilot solution with your own data using Microsoft Foundry](#)
- [Implement Retrieval Augmented Generation \(RAG\) with Azure OpenAI in Foundry Models](#)

Pluralsight training

- [Add search to apps \(Pluralsight\)](#)
- [Developer course \(Pluralsight\)](#)

Last updated on 11/18/2025