Azure AI Search provides a cloud-based solution for indexing and querying a wide range of data sources, and creating comprehensive and high-scale search solutions. With Azure AI Search, you can:

* Index documents and data from a range of sources.
* Use cognitive skills to enrich index data.
* Store extracted insights in a knowledge store for analysis and integration.

 **Creating Azure AI Search Solution**:

* Requires an Azure AI Search resource.
* May need additional Azure resources for data storage and application services.

 **Service Tiers and Capacity Management**:

* Must specify a pricing tier when creating a search resource.
* Pricing tiers determine capacity, configuration options, and cost.
* Available pricing tiers:
  + **Free (F)**: For exploration and tutorials.
  + **Basic (B)**: Small-scale solutions, max 15 indexes and 2 GB data.
  + **Standard (S)**: Enterprise-scale solutions, includes variants S, S2, S3 (increasing capacity) and S3HD (optimized for fast read performance).
  + **Storage Optimized (L)**: For large indexes, includes L1 and L2, higher query latency.
* Cannot change pricing tier after selection; must create a new resource to change tiers.

 **Replicas and Partitions**:

* **Replicas**: Instances of the search service, ensuring capacity for multiple concurrent queries and indexing.
* **Partitions**: Divide an index into multiple storage locations, splitting I/O operations.
* **Search Units (SU)**: Calculated as replicas multiplied by partitions (R x P = SU).
* Example: 4 replicas and 3 partitions equal 12 search units.

 **Data Source**:

* Starting point for the data to be searched.
* Supports multiple types:
  + Unstructured files in Azure blob storage.
  + Tables in Azure SQL Database.
  + Documents in Cosmos DB.
* Data can be pulled from these sources for indexing or pushed directly as JSON into an index.

 **Skillset**:

* Enhances basic search solutions by applying AI skills during the indexing process.
* Enrichment pipeline defined in a skillset.
* Examples of AI skills:
  + Detecting document language.
  + Extracting key phrases to identify main themes.
  + Sentiment analysis.
  + Recognizing specific entities (locations, people, organizations).
  + Generating descriptions of images or extracting text via OCR.
  + Custom skills for specific requirements.

 **Indexer**:

* Drives the indexing process.
* Maps data and metadata from the source and skillset outputs to index fields.
* Can run automatically, on a schedule, or on-demand.
* May need to reset the index when adding new fields or skills.

 **Index**:

* Searchable result of the indexing process.
* Collection of JSON documents with indexed fields.
* Attributes of index fields:
  + **key**: Unique identifier for records.
  + **searchable**: Enables full-text search.
  + **filterable**: Allows filtering based on field values.
  + **sortable**: Enables sorting of results.
  + **facetable**: Supports creating facets for filtering results.
  + **retrievable**: Fields included in search results (default unless removed).

 **Indexing Process Overview**:

* Creates a document for each indexed entity.
* Combines metadata from the data source with enriched fields from cognitive skills.
* Documents are JSON structures.

 **Initial Document Structure**:

* Begins with fields extracted directly from the source:
  + metadata\_storage\_name
  + metadata\_author
  + content

 **Handling Image Data**:

* Indexer can extract image data into a normalized\_images collection:
  + image0, image1, etc.
* Normalized images are used as inputs for skills extracting information from images.

 **Adding Skills Outputs**:

* Each skill adds fields to the document.
* Example: Language detection skill adds a language field.

 **Hierarchical Document Structure**:

* Skills are applied to specific contexts within the document's hierarchy.
* Example: OCR skill extracts text from each image in the normalized\_images collection.

 **Skill Outputs as Inputs**:

* Outputs from one skill can be inputs for subsequent skills.
* Example: A merge skill combines original text and image text into a merged\_content field.

 **Mapping Fields to Index**:

* Two ways to map fields in the final document to index fields:
  + **Direct Source Data Fields**:
    - Mapped to index fields either implicitly (automatically) or explicitly (defined mapping, renaming, or applying functions).
  + **Skill Output Fields**:
    - Explicitly mapped from their hierarchical location in the document to the target index field.

 **Example of Final Document Structure**:

* metadata\_storage\_name
* metadata\_author
* content
* normalized\_images
  + image0
    - Text
  + image1
    - Text
* language
* merged\_content
* **Querying an Index**:
  + Retrieve information from indexed document content.
  + Commonly uses full text search semantics.
* **Full Text Search**:
  + Parses text-based document contents to find query terms.
  + Based on Lucene query syntax with two variants:
    - **Simple**: Intuitive syntax for basic searches.
    - **Full**: Extended syntax for complex filtering and sophisticated queries.
* **Query Parameters**:
  + **search**: Search expression with terms to find.
  + **queryType**: Specifies Lucene syntax (simple or full).
  + **searchFields**: Index fields to search.
  + **select**: Fields to include in results.
  + **searchMode**: Criteria for results based on search terms (e.g., "Any" for documents containing any terms, "All" for documents containing all terms).
* **Query Processing Stages**:
  + **Query Parsing**:
    - Evaluates and reconstructs search expression into a tree of subqueries:
      * Term queries: Specific words (e.g., "hotel").
      * Phrase queries: Multi-term phrases in quotes (e.g., "free parking").
      * Prefix queries: Terms with specified prefix (e.g., air\*).
  + **Lexical Analysis**:
    - Analyzes and refines query terms based on linguistic rules:
      * Converts text to lower case.
      * Removes stopwords (e.g., "the", "a").
      * Converts words to root form (e.g., "comfortable" to "comfort").
      * Splits composite words into constituent terms.
  + **Document Retrieval**:
    - Matches query terms against indexed terms.
    - Identifies matching documents.
  + **Scoring**:
    - Assigns relevance scores to results using term frequency/inverse document frequency (TF/IDF) calculation.
* **Example Query Workflow**:
  + User searches for "comfortable hotel".
  + **Query Parsing**:
    - Identifies terms "comfortable" and "hotel".
  + **Lexical Analysis**:
    - Simplifies "comfortable" to "comfort".
  + **Document Retrieval**:
    - Finds documents containing "comfort" and "hotel".
  + **Scoring**:
    - Assigns scores based on relevance, showing most relevant results first.
* **Filtering Results**:
  + **Simple Search Expression**: Includes filter criteria directly.
  + **OData Filter Expression**: Uses $filter parameter with full syntax search expression.
  + Can apply filters to any filterable field in the index.

**Filtering with Facets**:

* Useful for fields with a small number of discrete values.
* Presents filtering criteria based on field values in the result set.
* Requires specifying facetable fields in the initial query.

**Sorting Results**:

* Default sort order is by relevancy score.
* Override default by using $orderby parameter.
* Specify sortable fields and sort order (asc or desc).

**Enhancing the Index**:

* **Search-as-you-type**:
  + **Suggestions**:
    - Retrieve and display a list of suggested results as the user types.
    - No need to submit the search query.
  + **Autocomplete**:
    - Completes partially typed search terms based on index field values.
  + **Implementation**:
    - Add a suggester to one or more fields in the index.
    - Use the suggestion and autocomplete REST API endpoints or .NET methods to retrieve suggested results or autocompleted terms.
  + **Example**:
    - Define a suggester in the index for fields like "productName" or "authorName".
    - Query with a partial term to get suggestions or autocompleted terms.
* **Custom Scoring and Result Boosting**:
  + **Default Scoring**:
    - Uses term-frequency/inverse-document-frequency (TF/IDF) algorithm.
  + **Custom Scoring Profiles**:
    - Apply weighting values to specific fields to increase the search score.
    - Boost results based on field values, such as recent modification date or file size.
  + **Implementation**:
    - Define a scoring profile in the index.
    - Specify the custom scoring profile in individual searches or set it as the default in the index definition.

**Synonyms**:

* **Purpose**:
  + Link related terms to help users find information referred to in different ways.
* **Synonym Maps**:
  + Define synonym relationships (e.g., "UK" <=> "United Kingdom" <=> "Great Britain").
  + Apply synonym maps to individual fields in the index.
* **Implementation**:
  + Create a synonym map and apply it to the index fields.

You can use the predefined skills in Azure AI Search to greatly enrich an index by extracting additional information from the source data. However, there may be occasions when you have specific data extraction needs that cannot be met with the predefined skills and require some custom functionality.

For example:

* Integrate the Form Recognizer service to extract data from forms
* Consume an Azure Machine Learning model to integrate predicted values into an index
* Any other custom logic

To support these scenarios, you can implement custom skills as web-hosted services (such as Azure Functions) that support the required interface for integration into a skillset.

Your custom skill must implement the expected schema for input and output data that is expected by skills in an Azure AI Search skillset.

To integrate a custom skill into your indexing solution, you must add a skill for it to a skillset using the **Custom.WebApiSkill** skill type.

The skill definition must:

* Specify the URI to your web API endpoint, including parameters and headers if necessary.
* Set the context to specify at which point in the document hierarchy the skill should be called
* Assign input values, usually from existing document fields
* Store output in a new field, optionally specifying a target field name (otherwise the output name is used)

 **Azure AI Search Overview**:

* Creates search solutions using AI skills to enrich data and populate an index.
* Enrichments add insights such as:
  + Language detection.
  + Key phrases.
  + Sentiment analysis.
  + Named entity recognition (locations, people, organizations, landmarks).
  + AI-generated image descriptions or text extracted via OCR.

 **Benefits of Enriched Data**:

* Provides a comprehensive search solution beyond basic full-text search.

 **Knowledge Stores**:

* **Purpose**:
  + Extends the utility of enriched data beyond the index.
  + Supports scenarios like data orchestration, analysis, and reporting.
* **Projections**:
  + JSON objects for data integration (e.g., with Azure Data Factory).
  + Relational schema for analysis/reporting (e.g., with Power BI).
  + Image files extracted from documents.
* **Implementation**:
  + Define a knowledge store in the skillset.
  + Indexer runs the enrichment pipeline and generates projections.
  + Projections are persisted in the knowledge store.

 **Use Cases**:

* Export JSON objects for integration with data orchestration tools.
* Normalize index records into tables for analytical tools.
* Save extracted images for further use.

To define the knowledge store and the projections you want to create in it, you must create a knowledgeStore object in the skillset that specifies the Azure Storage connection string for the storage account where you want to create projections, and the definitions of the projections themselves.

You can define object projections, table projections, and file projections depending on what you want to store; however note that you must define a separate projection for each type of projection, even though each projection contains lists for tables, objects, and files. Projection types are mutually exclusive in a projection definition, so only one of the projection type lists can be populated. If you create all three kinds of projection, you must include a projection for each type

Language Studio is included with Azure AI Language. Language Studio lets you use UI tools to explore and build AI models focused in language into your solutions.

These AI models have endpoints that can be used in search solutions to enrich indexes.

**Explore the available features of Azure AI Language**

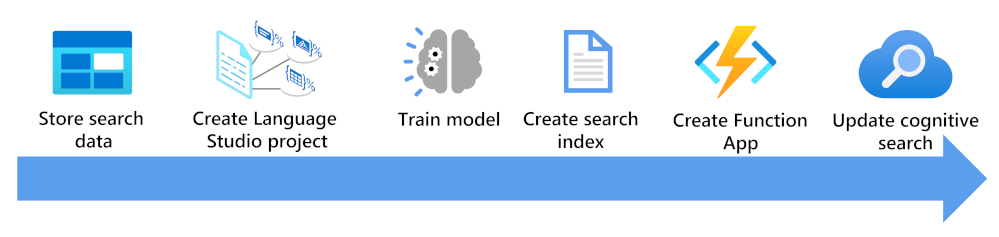
Here you'll explore the features Azure AI Language offers and then use the demo-like environment to test a preconfigured sentiment model. You'll then see the steps to create, train and deploy a custom model for conversational language understanding.

**Azure AI Language features**

Azure AI Language groups its features into the following areas:

1. Classify text
2. Understand questions and conversational language
3. Extract information
4. Summarize text
5. Translate text

**Enriching a Search Index in Azure AI Search with Custom Classes and Azure AI Language**



* **Store Your Documents**:
  + Use Azure Blob storage accessible by Language Studio and Azure AI Search indexers.
  + Ensure container accessibility (can use Container or private containers with additional configuration).
* **Create a Custom Text Classification Project**:
  + Choose appropriate classification type:
    - **Single Label**: Documents map to a single class.
    - **Multi Label**: Documents can map to multiple classes.
  + Classify documents manually using Language Studio's graphical tool or label all documents before project creation.
* **Train and Test Your Model**:
  + Utilize Language Studio to train and test your custom text classification model.
* **Create a Search Index**:
  + Define a search index based on stored documents.
  + Configure index fields and mappings according to classification needs.
* **Create a Function App**:
  + Develop a function app to deploy and use your trained model.
* **Update Your Search Solution**:
  + Update index, indexer, and custom skillset in your Azure AI Search solution:
    - Define mappings between classified documents and search index fields.
    - Configure indexers to access and retrieve data from Blob storage.
    - Incorporate custom skillset to apply classification results to index documents.

Azure AI Search is a powerful search service that can index a wide range of data from various sources. A core part of search is returning relevant results from search queries.

To improve the ranking of a document with term boosting in Azure AI Search, you can utilize the Lucene query syntax to prioritize specific terms in your search query. Here's how you can do it:

1. **Write a Simple Query**:
   * Initially, you might start with a simple query like:

bash

Copy code

search=luxury&$select=HotelId, HotelName, Category, Tags, Description&$count=true

* + This query searches for the term "luxury" across all fields in the documents.

1. **Enable the Lucene Query Parser**:
   * By default, Azure AI Search uses a simple syntax for queries. You can enable the Lucene Query Parser for more advanced query capabilities.
   * Example query with Lucene syntax:

sql

Copy code

search=luxury AND air con&$select=HotelId, HotelName, Category, Tags, Description&$count=true&queryType=full

* + With Lucene syntax, you can use boolean operators, fielded search, fuzzy search, term proximity search, regular expression search, wildcard search, precedence grouping, and term boosting.

1. **Boost Search Terms**:
   * To prioritize specific terms in your query, you can use term boosting.
   * For example, to prioritize hotels in the "luxury" category and search for "air con" in the "Tags" field:

ruby

Copy code

(Description:luxury OR Category:luxury^3) AND Tags:'air con'\*

* + This query assigns a higher score to hotels with "luxury" in the description or category field. The "^3" boosts the importance of the "luxury" category.
  + Add other query string parameters for selectivity and result count.

ruby

Copy code

search=(Description:luxury OR Category:luxury^3) AND Tags:'air con'\*&$select=HotelId, HotelName, Category, Tags, Description&$count=true&queryType=full

By incorporating term boosting into your search queries, you can ensure that the most relevant documents are ranked higher in the search results, leading to a better user experience.

To improve the relevance of search results in Azure AI Search, you can add scoring profiles, which allow you to influence the document scores based on your own criteria. Here's how you can do it:

1. **Understand How Search Scores are Calculated**:
   * Azure AI Search uses the BM25 similarity ranking algorithm to score documents based on the search terms used.
   * The score is calculated based on factors such as the frequency of terms in a document, the document's size, and the rarity of each term.
   * By default, search results are ordered by their search score, with the highest scoring documents appearing first.
2. **Improve the Score for More Relevant Documents**:
   * As the default scoring may not always return the highest score for the most relevant document, you can use scoring profiles to influence document scores.
   * Scoring profiles allow you to define different weights for fields in an index, making certain fields more relevant than others.
   * Additionally, scoring profiles can include functions such as distance or freshness, providing more control over how documents are scored.
3. **Add a Weighted Scoring Profile**:
   * You can add scoring profiles to a search index in the Azure portal.
   * Navigate to your search service, select the index, and then select "Scoring profiles."
   * Click on "+ Add scoring profile" and specify a unique name for the profile.
   * Choose the field you want to assign a weight to, and enter the weight value.
   * Save the scoring profile, and optionally set it as the default profile to be applied to all searches.
4. **Use Functions in a Scoring Profile**:
   * Scoring profiles can also include functions such as Magnitude, Freshness, Distance, or Tag, allowing you to further customize how scores are calculated.
   * For example, you can use the Magnitude function to alter scores based on the range of values for a numeric field like "Rating."

By adding scoring profiles to your Azure AI Search solution, you can ensure that the most relevant documents are ranked higher in the search results, leading to a better user experience.

Custom analyzers in Azure AI Search allow you to control how the content of a field is processed and split into tokens for inclusion in the index. Let's break down the components of a custom analyzer:

1. **Character Filters**:
   * Character filters process a string before it reaches the tokenizer.
   * Available character filters include:
     + html\_strip: Removes HTML constructs such as tags and attributes.
     + mapping: Allows you to specify mappings to replace one string with another.
     + pattern\_replace: Enables you to specify regular expressions to identify patterns in the input text and replace matching text.
2. **Tokenizers**:
   * Tokenizers divide the text into tokens that will be stored in the index and often break down words into their root forms.
   * There are various tokenizers available, such as:
     + classic: Processes text based on grammar for European languages.
     + keyword: Emits the entire input as a single token.
     + lowercase: Divides text at non-letters and converts resulting tokens to lowercase.
     + whitespace: Divides text wherever there's white space.
3. **Token Filters**:
   * Token filters execute additional processing on tokens emitted by the tokenizer.
   * Available token filters include:
     + Language-specific filters like arabic\_normalization and classic which apply language-specific grammar rules.
     + apostrophe: Removes any apostrophes from tokens and characters after the apostrophe.
     + keep: Removes tokens that don't include one or more words from a specified list.
     + length: Removes tokens based on their length compared to specified minimum and maximum values.
     + trim: Removes leading and trailing white space from tokens.

By combining these components, you can create custom analyzers tailored to your specific requirements. Whether you need to handle unusual text formats, remove specific characters, or apply language-specific processing, custom analyzers provide the flexibility to fine-tune how your data is indexed and searched in Azure AI Search.

Support for multiple languages can be added to a search index. You can add language support manually by providing all the translated text fields in all the different languages you want to support. You could also choose to use Azure AI Services to provide translated text through an enrichment pipeline.

Geo-spatial functions in Azure AI Search allow users to search for items based on their geographical location. Two key functions for this purpose are geo.distance and geo.intersects.

1. **geo.distance Function**:
   * Calculates the distance in kilometers between two points on the Earth's surface.
   * Syntax: geo.distance(field\_name, geography'POINT(longitude latitude)').

**geo.intersects Function**:

* Checks if a location falls within a specified polygon on the Earth's surface.
* Syntax: geo.intersects(field\_name, geography'POLYGON((longitude1 latitude1, longitude2 latitude2, ...))').

By leveraging these geo-spatial functions, users can easily find items based on their proximity to a reference point or within a defined geographical area, enhancing the search experience for location-based queries.

Azure Machine Learning allows you to build powerful AI models and host them on Azure. These AI models have endpoints that can be used in search solutions to enrich indexes.

Using a machine learning custom skill works the same as adding any other custom skill to a search index.

When you enrich a search index with an Azure Machine Learning (AML) custom skill, the enrichment happens at the document level. The skillset used by your document indexer needs to include an AmlSkill.

Take note that the custom skill doesn't include settings for batchSize. As the AML model will process a single document at a time. The remaining settings that control the performance of the skill are timeout and degreeOfParallelism. The above schema has set 30 seconds as the timeout value. The degree of parallelism should start at one. Depending on your infrastructure, you might be able to increase this number.

The best way to manage the efficiency of an AML skill is to scale up the Kubernetes inference cluster appropriately to manage your workload.

The index for the document needs a field to store the results from the AML model. You'll then add an output field mapping to store the results from the custom skill set to the field on the document in the index.

To enrich a search index using an Azure Machine Learning (AML) model, you can follow these steps:

1. **Create an AML Workspace**:
   * Set up an AML workspace, which includes resources like storage accounts and key stores. You can do this in the Azure portal, and it provides you with a link to launch the Azure AI Machine Learning Studio.
2. **Create and Train a Model in Azure Machine Learning Studio**:
   * Use the Azure AI Machine Learning Studio to create and train models. You can utilize prebuilt components or design pipelines using drag-and-drop functionality. Once the model is trained, it needs to be registered in Azure AI Machine Learning Studio for deployment.
3. **Alter Model Behavior for AML Custom Skill**:
   * Modify the model code to handle single rows of data instead of entire datasets. Ensure that the JSON response from the model contains only the output prediction.
4. **Create an Endpoint for the Model**:
   * Deploy the model to an endpoint. Azure AI Machine Learning Studio supports deploying models to real-time, batch, or web service endpoints. However, for use with the AML custom skill in Azure AI Search, the endpoint must be a web service endpoint hosted on Azure Kubernetes Service (AKS).
5. **Connect AML Custom Skill to the Endpoint**:
   * Add a new field to your search index to include the output from the AML model. Update your index skillset by adding the #Microsoft.Skills.Custom.AmlSkill custom skill. Modify your indexer to map the output from the custom skill to the newly created field in the index. Rerun the indexer to enrich your index with the AML model.

Azure AI Search supports two main ways to get data into a search index. So far, you've seen how to pull data using indexers. The most flexible way to get data into an index is to push the data into it.

To push data from an external source into a search index using Azure Data Factory (ADF), follow these steps:

1. **Create an Azure AI Search Index**:
   * Set up an Azure AI Search service and create an index with all the fields you want to store data in.
2. **Create a Pipeline in Azure Data Factory**:
   * Design a pipeline with a copy data activity to push data into the search index.
3. **Configure Data Source Connection**:
   * Create a data source connection to where your data resides. In this case, use the HTTP connector to connect to the JSON file.
4. **Configure Sink Connection**:
   * Create a sink connection to your Azure AI Search index. Select the Azure Search connector and specify your Azure subscription and search service.
5. **Map Source Fields to Target Fields**:
   * Map the fields from your source data to your search index. If the field names in your JSON document match the field names in the index, ADF will automatically map them.
6. **Run the Pipeline**:
   * Validate and deploy the pipeline, then run it to push the data into the index. Once the pipeline is executed, the JSON data will be added to your search index.

It's important to note that the Azure AI Search linked service in ADF has limitations. Currently, it only supports certain data types like String, Int32, Int64, Double, Boolean, and DateTimeOffset. Complex types and arrays are not supported, so only the first telephone number from the JSON document in the example will be mapped to the search index.

Running a successful Azure AI Search solution requires you to understand how to maintain its two primary workloads of indexing and querying. It's essential that the search solution is as cost effective as possible.

Securing your Azure AI Search solution is essential to maintain the integrity and confidentiality of your data. Let's break down the key aspects of securing your search solution:

1. **Data Encryption**:
   * Azure AI Search encrypts data at rest using service-managed keys, ensuring that your indexes, data sources, and other components are encrypted.
   * Data in transit is encrypted using HTTPS TLS 1.3 encryption over port 443, providing secure communication between clients and the search service.
   * Optionally, you can use customer-managed keys stored in Azure Key Vault for additional control over encryption keys.
2. **Secure Inbound Traffic**:
   * Restrict access to your search solution's public endpoint using firewalls to allow access only from specific IP addresses.
   * For heightened security, consider using ExpressRoute, Azure Gateway, and Azure private link to control inbound traffic and limit exposure to the internet.
3. **Authentication**:
   * Authenticate inbound search requests using key-based authentication. Azure AI Search provides admin keys for write permissions and query keys for read permissions.
   * Role-based access control (RBAC) allows you to define roles with specific permissions for administering the service or managing data plane operations.
4. **Secure Outbound Traffic**:
   * Secure outbound connections from your search solution to other servers using key-based authentication, database logins, or managed identities.
   * Use firewalls to restrict outbound traffic, allowing connections only to trusted endpoints. Consider using Azure private link for secure connectivity to Azure services.
5. **Document-Level Security**:
   * Implement document-level security to restrict access to specific documents based on user or group permissions.
   * Add a security field to each document containing user or group IDs with access rights. Filter search results based on these permissions using the search.in filter in your queries.

The default option when you create your ACS is key-based authentication. There are two different kinds of keys:

* **Admin keys** - grant your write permissions and the right to query system information (*maximum of 2 admin keys can be created per search service*)
* **Query keys** - grant read permissions and are used by your users or apps to query indexes (*maximum of 50 query keys can be created per search service*)

The built-in roles you can assign to manage the Azure AI Search service are:

* **Owner** - Full access to all search resources
* **Contributor** - Same as above, but without the ability to assign roles or change authorizations
* **Reader** - View partial service information

If you need a role that can also manage the data plane for example search indexes or data sources, use one of these roles:

* **Search Service Contributor** - A role for your search service administrators (the same access as the Contributor role above) and the content (indexes, indexers, data sources, and skillsets)
* **Search Index Data Contributor** - A role for developers or index owners who will import, refresh, or query the documents collection of an index
* **Search Index Data Reader** - Read-only access role for apps and users who only need to run queries

**Measure Current Search Performance**:

* + Enable diagnostic logging in Azure portal.
  + Capture diagnostic information at the search service level to identify performance bottlenecks.
  + Monitor for throttling using HTTP response codes (503 for searches, 207 for indexes).
* **Check Performance of Individual Queries**:
  + Use tools like Postman to test query performance.
  + Analyze response headers for elapsed time values to measure query completion time.
* **Optimize Index Size and Schema**:
  + Review and remove irrelevant documents from the index.
  + Simplify schema by reducing complexity, unnecessary fields, and attributes.
  + Consider the storage limitations of attributes on fields to optimize index size.
* **Improve Query Performance**:
  + Specify only necessary fields using searchFields parameter.
  + Minimize returned fields to reduce response time.
  + Avoid partial search terms and high skip values.
  + Limit the use of facetable and filterable fields to low cardinality data.
  + Use search functions instead of individual values in filter criteria.
* **Scale Your Search Service**:
  + Scale out by adding more partitions to distribute the index across multiple storage units.
  + Choose the appropriate service tier based on performance requirements, considering storage, replicas, and partitions.
  + Plan for future growth by considering search units (SU), which are the product of replicas and partitions.
* **Choose the Best Service Tier**:
  + Consider the trade-offs between scalability, performance, and cost for each service tier.
  + Evaluate the performance benefits of scaling out for parallelism versus the additional resources and capabilities offered by higher tiers.
  + Plan for future scalability needs by understanding the relationship between replicas, partitions, and search units.

 **Improve Reliability**:

* Increase the number of replicas to enhance availability:
  + Two replicas guarantee 99.9% availability for queries.
  + Three or more replicas ensure 99.9% availability for both queries and indexing.
* Utilize Availability Zones for redundancy:
  + Host replicas in different Availability Zones to mitigate risks of data center failures.

 **Distribute Search Solution Globally**:

* Host multiple instances of search services in different geographical regions for improved availability and performance.
* Benefits include protection against regional failures and reduced response times for globally distributed users or apps.
* Replicate indexes across regions using indexers or the Push API, and manage search requests through Azure Traffic Manager for optimal routing.

 **Backup Options for Search Indexes**:

* While Azure AI Search lacks a formal backup and restore mechanism, you can create custom tools to back up index definitions as JSON files.
* These files can be used to recreate search indexes in case of any unforeseen issues or disasters.

Semantic ranking in Azure AI Search is a feature designed to enhance the relevance of search results by leveraging language understanding. Here's a breakdown of its key aspects:

* **BM25 Ranking Function**:
  + Azure AI Search primarily uses the BM25 ranking function to rank search results based on the frequency of search terms within documents.
* **Semantic Ranking**:
  + Semantic ranking supplements BM25 by incorporating language understanding to improve result relevance.
  + It calculates a new relevance score using BM25 results combined with language understanding models to extract query context and meaning.
* **Semantic Captions and Answers**:
  + Semantic ranking provides additional information alongside ranked results:
    - **Semantic Captions**: Summarize document content and highlight relevant text.
    - **Semantic Answers**: Optional feature providing answers to questions found within search results.
* **Working Process**:
  + Semantic ranking takes the top 50 results from BM25 ranking, converts them into text strings, and processes them using machine reading comprehension models.
  + It then ranks semantic captions based on relevance and returns them in descending order.
* **Advantages**:
  + Enhances result relevance by closely matching query semantics.
  + Provides captions and answers to improve user understanding.
* **Limitations**:
  + Applies to results returned by BM25; won't provide additional documents.
  + Considers only the top 50 results from BM25.
* **Pricing**:
  + Offers up to 1000 free semantic ranking queries per month.
  + Standard pricing applies for more than 1000 queries, based on search volume, type, and region.

Semantic ranking thus enhances search result quality by incorporating language understanding, offering additional context, and improving relevance, albeit within certain limitations and pricing considerations.