

- Assign roles to groups instead of individual users to reduce administrative overhead and errors in managing individual roles.
- Assign the service principal at the root collection for automation purposes.
- To increase security, enable Microsoft Entra Conditional Access with multifactor authentication for at least Collection Admins, Data Source Admins, and Data Curators. Make sure emergency accounts are excluded from the Conditional Access policy.

Design considerations

- Microsoft Purview access management has moved into data plane. Azure Resource Manager roles aren't used anymore, so you should use Microsoft Purview to assign roles.
- In Microsoft Purview, you can assign roles to users, security groups, and service principals (including managed identities) from Microsoft Entra ID on the same Microsoft Entra tenant where the Microsoft Purview account is deployed.
- You must first add guest accounts to your Microsoft Entra tenant as B2B users before you can assign Microsoft Purview roles to external users.
- By default, Collection Admins don't have access to read or modify assets. But they can elevate their access and add themselves to more roles.
- By default, all role assignments are automatically inherited by all child collections. But you can enable **Restrict inherited permissions** on any collection except for the root collection. **Restrict inherited permissions** removes the inherited roles from all parent collections, except for the Collection Admin role.
- For Azure Data Factory connection: to connect to Azure Data Factory, you have to be a Collection Admin for the root collection.
- If you need to connect to Azure Data Factory for lineage, grant the Data Curator role to the data factory's managed identity at your Microsoft Purview root collection level. When you connect Data Factory to Microsoft Purview in the authoring UI, Data Factory tries to add these role assignments automatically. If you have the Collection Admin role on the Microsoft Purview root collection, this operation will work.

Collections archetypes

You can deploy your Microsoft Purview collection based on centralized, decentralized, or hybrid data management and governance models. Base this decision on your business and security requirements.

Example 1: Single-region organization

This structure is suitable for organizations that:

- Are mainly based in a single geographic location.
- Have a centralized data management and governance team where the next level of data management falls into departments, teams, or projects.

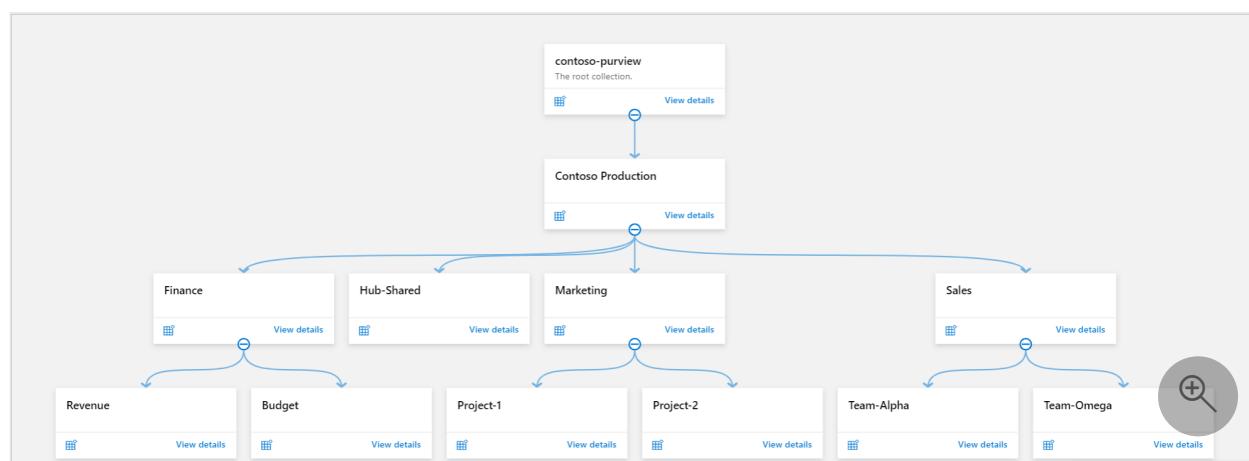
The collection hierarchy consists of these verticals:

- Root collection (default)
- Contoso (top-level collection)
- Departments (a delegated collection for each department)
- Teams or projects (further segregation based on projects)

Each data source is registered and scanned in its corresponding collection. So assets also appear in the same collection.

Organization-level shared data sources are registered and scanned in the Hub-Shared collection.

The department-level shared data sources are registered and scanned in the department collections.



Example 2: Multiregion organization

This scenario is useful for organizations:

- That have a presence in multiple regions.

- Where the data governance team is centralized or decentralized in each region.
- Where data management teams are distributed in each geographic location.

The collection hierarchy consists of these verticals:

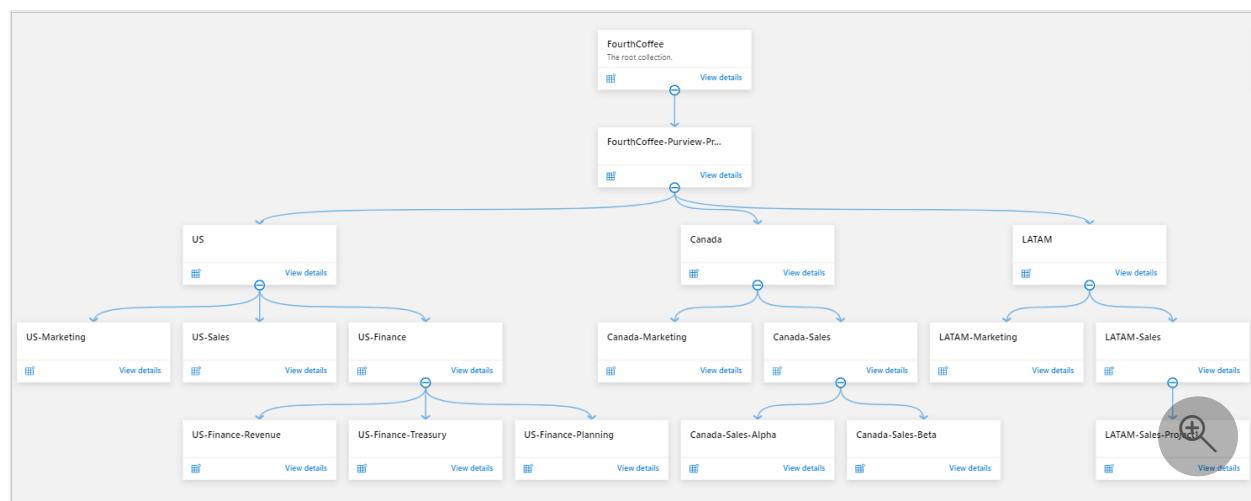
- Root collection (default)
- FourthCoffee (top-level collection)
- Geographic locations (mid-level collections based on geographic locations where data sources and data owners are located)
- Departments (a delegated collection for each department)
- Teams or projects (further segregation based on teams or projects)

In this scenario, each region has a subcollection of its own under the top-level collection in the Microsoft Purview account. Data sources are registered and scanned in the corresponding subcollections in their own geographic locations. So assets also appear in the subcollection hierarchy for the region.

If you have centralized data management and governance teams, you can grant them access from the top-level collection. When you do, they gain oversight for the entire data estate in the data map. Optionally, the centralized team can register and scan any shared data sources.

Region-based data management and governance teams can obtain access from their corresponding collections at a lower level.

The department-level shared data sources are registered and scanned in the department collections.



Example 3: Multiregion, data transformation

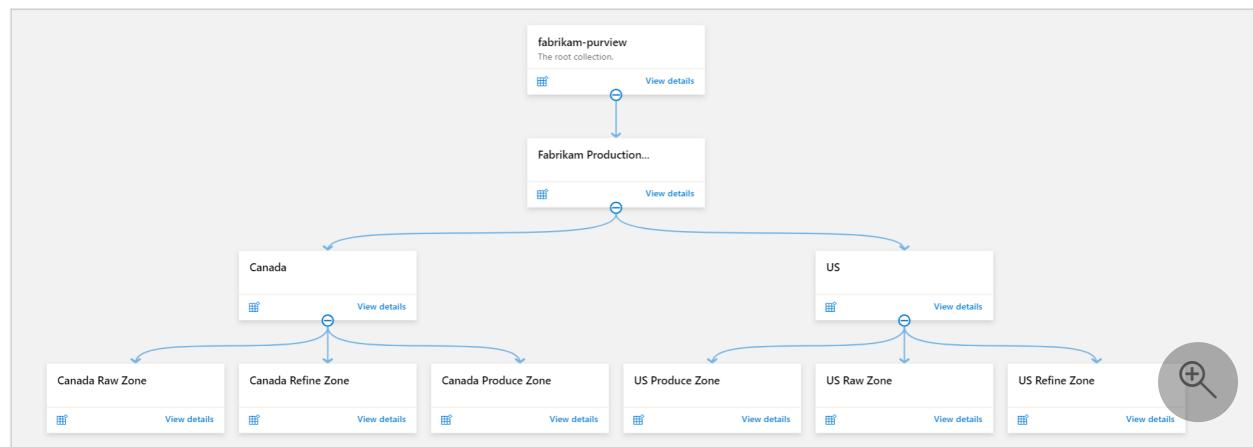
This scenario can be useful if you want to distribute metadata-access management based on geographic locations and data transformation states. Data scientists and data

engineers who can transform data to make it more meaningful can manage Raw and Refine zones. They can then move the data into Produce or Curated zones.

The collection hierarchy consists of these verticals:

- Root collection (default)
- Fabrikam (top-level collection)
- Geographic locations (mid-level collections based on geographic locations where data sources and data owners are located)
- Data transformation stages (Raw, Refine, Produce/Curated)

Data scientists and data engineers can have the Data Curators role on their corresponding zones so they can curate metadata. Data Reader access to the curated zone can be granted to entire data personas and business users.



Example 4: Multiregion, business functions

This option can be used by organizations that need to organize metadata and access management based on business functions.

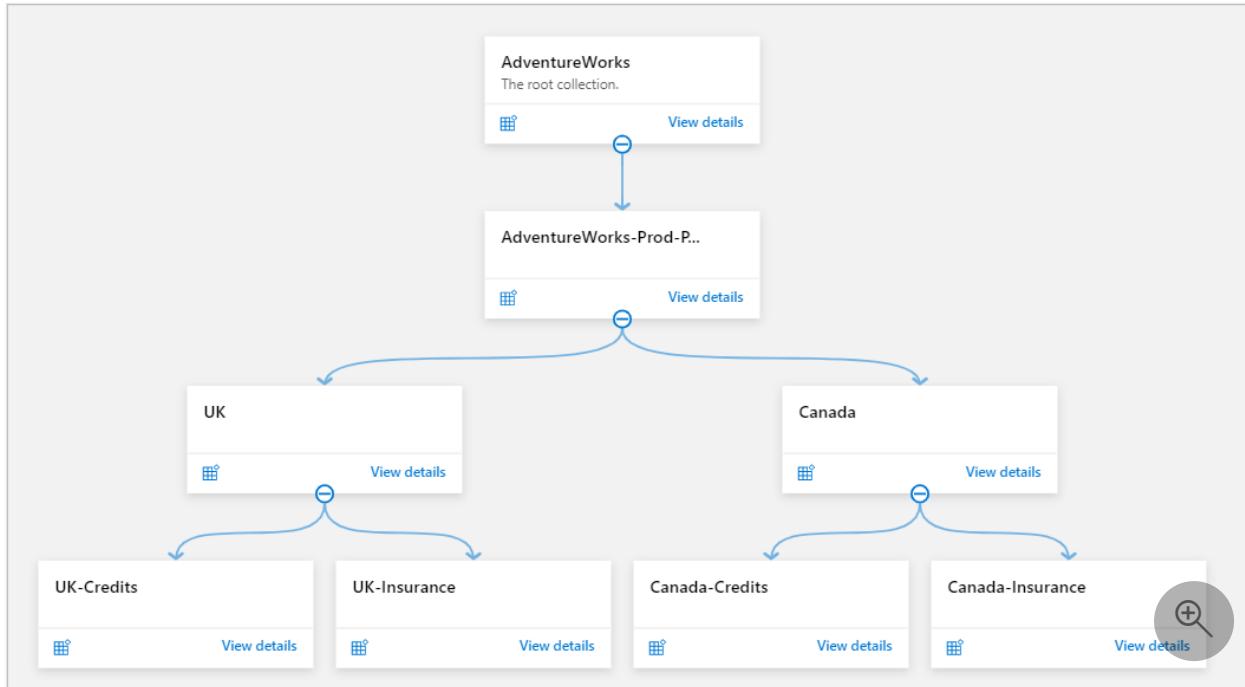
The collection hierarchy consists of these verticals:

- Root collection (default)
- AdventureWorks (top-level collection)
- Geographic locations (mid-level collections based on geographic locations where data sources and data owners are located)
- Major business functions or clients (further segregation based on functions or clients)

Each region has a subcollection of its own under the top-level collection in the Microsoft Purview account. Data sources are registered and scanned in the corresponding subcollections in their own geographic locations. So assets are added to the subcollection hierarchy for the region.

If you have centralized data management and governance teams, you can grant them access from the top-level collection. When you do, they gain oversight for the entire data estate in the data map. Optionally, the centralized team can register and scan any shared data sources.

Region-based data management and governance teams can obtain access from their corresponding collections at a lower level. Each business unit has its own subcollection.



Access management options

If you want to implement data democratization across an entire organization, assign the Data Reader role at the top-level collection to data management, governance, and business users. Assign Data Source Admin and Data Curator roles at the subcollection levels to the corresponding data management and governance teams.

If you need to restrict access to metadata search and discovery in your organization, assign Data Reader and Data Curator roles at the specific collection level. For example, you could restrict US employees so they can read data only at the US collection level and not in the LATAM collection.

You can apply a combination of these two scenarios in your Microsoft Purview data map if total data democratization is required with a few exceptions for some collections. You can assign Microsoft Purview roles at the top-level collection and restrict inheritance to the specific child collections.

Assign the Collection Admin role to the centralized data security and management team at the top-level collection. Delegate further collection management of lower-level

collections to corresponding teams.

Next steps

- [Create a collection and assign permissions in Microsoft Purview](#)
- [Create and manage collections in Microsoft Purview](#)
- [Access control in Microsoft Purview](#)

Microsoft Purview (formerly Azure Purview) deployment best practices

Article • 08/31/2023

This article is a guide to successfully deploying Microsoft Purview (formerly Azure Purview) into production in your data estate. It's intended to help you strategize and phase your deployment from research to hardening your production environment, and is best used in tandem with our [deployment checklist](#).

Note

These best practices cover the deployment for **Microsoft Purview unified governance solutions**. For more information about Microsoft Purview risk and compliance solutions, [go here](#). For more information about Microsoft Purview in general, [go here](#).

If you're looking for a strictly technical deployment guide, use the [deployment checklist](#).

If you're creating a plan to deploy Microsoft Purview and want to consider best practices as you develop your deployment strategy, then follow the article below. This guide outlines tasks can be completed in phases over the course of a month or more to develop your deployment process for Microsoft Purview. Even organizations who have already deployed Microsoft Purview can use this guide to ensure they're getting the most out of their investment.

A well-planned deployment of your governance platform, can give the following benefits:

- Better data discovery
- Improved analytic collaboration
- Maximized return on investment

This guide provides insight on a full deployment lifecycle, from initial planning to a mature environment by following these stages:

 [Expand table](#)

Stage	Description
Identify objectives and goals	Consider what your entire organization wants and needs from data governance.

Stage	Description
Gathering questions	What questions might you and your team have as you get started, and where can you look to begin addressing them?
Create a process to move to production	Create a phased deployment strategy tailored to your organization.
Platform hardening	Continue to grow your deployment to maturity.

Many of Microsoft Purview's applications and features have their own individual best practices pages as well. They're referenced often throughout this deployment guide, but you can find all of them in the table of contents under **Concepts** and then **Best practices and guidelines**.

Identify objectives and goals

Many organizations have started their data governance journey by developing individual solutions that cater to specific requirements of isolated groups and data domains across the organization. Although experiences may vary depending on the industry, product, and culture, most organizations find it difficult to maintain consistent controls and policies for these types of solutions.

Some of the common data governance objectives that you might want to identify in the early phases to create a comprehensive data governance experience include:

- Maximizing the business value of your data
- Enabling a data culture where data consumers can easily find, interpret, and trust data
- Increasing collaboration amongst various business units to provide a consistent data experience
- Fostering innovation by accelerating data analytics to reap the benefits of the cloud
- Decreasing time to discover data through self-service options for various skill groups
- Reducing time-to-market for the delivery of analytics solutions that improve service to their customers
- Reducing the operational risks that are due to the use of domain-specific tools and unsupported technology

The general approach is to break down those overarching objectives into various categories and goals. Some examples are:

Category	Goal
Discovery	Admin users should be able to scan Azure and non-Azure data sources (including on-premises sources) to gather information about the data assets automatically.
Classification	The platform should automatically classify data based on a sampling of the data and allow manual override using custom classifications.
Consumption	The business users should be able to find information about each asset for both business and technical metadata.
Lineage	Each asset must show a graphical view of underlying datasets so that the users understand the original sources and what changes have been made.
Collaboration	The platform must allow users to collaborate by providing additional information about each data asset.
Reporting	The users must be able to view reporting on the data estate including sensitive data and data that needs extra enrichment.
Data governance	The platform must allow the admin to define policies for access control and automatically enforce the data access based on each user.
Workflow	The platform must have the ability to create and modify workflow so that it's easy to scale out and automate various tasks within the platform.
Integration	Other third-party technologies such as ticketing or orchestration must be able to integrate into the platform via script or REST APIs.

Identify key scenarios

Microsoft Purview governance services can be used to centrally manage data governance across an organization's data estate spanning cloud and on-premises environments. To have a successful implementation, you must identify key scenarios that are critical to the business. These scenarios can cross business unit boundaries or affect multiple user personas either upstream or downstream.

These scenarios can be written up in various ways, but you should include at least these five dimensions:

1. Persona – Who are the users?
2. Source system – What are the data sources such as Azure Data Lake Storage Gen2 or Azure SQL Database?
3. Impact Area – What is the category of this scenario?
4. Detail scenarios – How the users use Microsoft Purview to solve problems?

5. Expected outcome – What is the success criteria?

The scenarios must be specific, actionable, and executable with measurable results. Some example scenarios that you can use:

[Expand table](#)

Scenario	Detail	Persona
Catalog business-critical assets	I need to have information about each data sets to have a good understanding of what it is. This scenario includes both business and technical metadata data about the data set in the catalog. The data sources include Azure Data Lake Storage Gen2, Azure Synapse DW, and/or Power BI. This scenario also includes on-premises resources such as SQL Server.	Business Analyst, Data Scientist, Data Engineer
Discover business-critical assets	I need to have a search engine that can search through all metadata in the catalog. I should be able to search using technical term, business term with either simple or complex search using wildcard.	Business Analyst, Data Scientist, Data Engineer, Data Admin
Track data to understand its origin and troubleshoot data issues	I need to have data lineage to track data in reports, predictions, or models back to its original source. I also need to understand the changes made to the data, and where the data has resided throughout the data life cycle. This scenario needs to support prioritized data pipelines Azure Data Factory and Databricks.	Data Engineer, Data Scientist
Enrich metadata on critical data assets	I need to enrich the data set in the catalog with technical metadata that is generated automatically. Classification and labeling are some examples.	Data Engineer, Domain/Business Owner
Govern data assets with friendly user experience	I need to have a Business glossary for business-specific metadata. The business users can use Microsoft Purview for self-service scenarios to annotate their data and enable the data to be discovered easily via search.	Domain/Business Owner, Business Analyst, Data Scientist, Data Engineer

Integration points with Microsoft Purview

It's likely that a mature organization already has an existing data catalog. The key question is whether to continue to use the existing technology and sync with the Microsoft Purview Data Map and Data Catalog or not. To handle syncing with existing

products in an organization, [Microsoft Purview provides Atlas REST APIs](#). Atlas APIs provide a powerful and flexible mechanism handling both push and pull scenarios. Information can be published to Microsoft Purview using Atlas APIs for bootstrapping or to push latest updates from another system into Microsoft Purview. The information available in Microsoft Purview can also be read using Atlas APIs and then synced back to existing products.

For other integration scenarios such as ticketing, custom user interface, and orchestration you can use Atlas APIs and Kafka endpoints. In general, there are four integration points with Microsoft Purview:

- **Data Asset** – This enables Microsoft Purview to scan a store's assets in order to enumerate what those assets are and collect any readily available metadata about them. So for SQL this could be a list of DBs, tables, stored procedures, views and config data about them kept in places like `sys.tables`. For something like Azure Data Factory (ADF) this could be enumerating all the pipelines and getting data on when they were created, last run, current state.
- **Lineage** – This enables Microsoft Purview to collect information from an analysis/data mutation system on how data is moving around. For something like Spark this could be gathering information from the execution of a notebook to see what data the notebook ingested, how it transformed it and where it outputted it. For something like SQL, it could be analyzing query logs to reverse engineer what mutation operations were executed and what they did. We support both push and pull based lineage depending on the needs.
- **Classification** – This enables Microsoft Purview to take physical samples from data sources and run them through our classification system. The classification system figures out the semantics of a piece of data. For example, we may know that a file is a Parquet file and has three columns and the third one is a string. But the classifiers we run on the samples will tell us that the string is a name, address, or phone number. Lighting up this integration point means that we've defined how Microsoft Purview can open up objects like notebooks, pipelines, parquet files, tables, and containers.
- **Embedded Experience** – Products that have a "studio" like experience (such as ADF, Synapse, SQL Studio, PBI, and Dynamics) usually want to enable users to discover data they want to interact with and also find places to output data. Microsoft Purview's catalog can help to accelerate these experiences by providing an embedding experience. This experience can occur at the API or the UX level at the partner's option. By embedding a call to Microsoft Purview, the organization can take advantage of Microsoft Purview's map of the data estate to find data assets, see lineage, check schemas, look at ratings, contacts etc.

Gathering questions

Once your organization agrees on the high-level objectives and goals, there will be many questions from multiple groups. It's crucial to gather these questions in order to craft a plan to address all of the concerns. Make sure to [include relevant groups](#) as you gather these questions. You can use our documentation to start answering them.

Some example questions that you may run into during the initial phase:

- What are the main data sources and data systems in our organization?
- [What data sources are supported?](#)
- For data sources that aren't supported yet by Microsoft Purview, what are my options?
- [How should we budget for Microsoft Purview?](#)
- [Who will use Microsoft Purview, and what roles will they have?](#)
- [Who can scan new data sources?](#)
- [Who can modify content inside of Microsoft Purview?](#)
- What processes can I use to improve the data quality in Microsoft Purview?
- How to bootstrap the platform with existing critical assets, [glossary terms](#), and contacts?
- [How can we secure Microsoft Purview?](#)
- How can we gather feedback and build a sustainable process?
- [What can we do in a disaster situation?](#)
- [We're already using Azure Data Catalog, can we migrate to Microsoft Purview?](#)

Even if you might not have the answer to most of these questions right away, gathering questions can help your organization to frame this project and ensure all "must-have" requirements can be met.

Include the right stakeholders

To ensure the success of implementing Microsoft Purview for your entire organization, it's important to involve the right stakeholders. Only a few people are involved in the initial phase. However, as the scope expands, you'll require more personas to contribute to the project and provide feedback.

Some key stakeholders that you may want to include:

[] [Expand table](#)

Persona	Roles
Chief Data Officer	The CDO oversees a range of functions that may include data management, data quality, master data management, data science, business intelligence, and creating data strategy. They can be the sponsor of the Microsoft Purview implementation project.
Domain/Business Owner	A business person who influences usage of tools and has budget control
Data Analyst	Able to frame a business problem and analyze data to help leaders make business decisions
Data Architect	Design databases for mission-critical line-of-business apps along with designing and implementing data security
Data Engineer	Operate and maintain the data stack, pull data from different sources, integrate and prepare data, set up data pipelines
Data Scientist	Build analytical models and set up data products to be accessed by APIs
DB Admin	Own, track, and resolve database-related incidents and requests within service-level agreements (SLAs); May set up data pipelines
DevOps	Line-of-Business application development and implementation; may include writing scripts and orchestration capabilities
Data Security Specialist	Assess overall network and data security, which involves data coming in and out of Microsoft Purview

Create a process to move to production

Below we've provided a potential four phase deployment plan that includes tasks, helpful links, and acceptance criteria for each phase:

1. [Phase 1: Pilot](#)
2. [Phase 2: Minimum viable product](#)
3. [Phase 3: Pre-production](#)
4. [Phase 4: Production](#)

Phase 1: Pilot

In this phase, Microsoft Purview must be created and configured for a small set of users. Usually, it's just a group of 2-3 people working together to run through end-to-end scenarios. They're considered the advocates of Microsoft Purview in their organization.

The main goal of this phase is to ensure key functionalities can be met and the right stakeholders are aware of the project.

Tasks to complete

 Expand table

Task	Detail	Duration
Gather & agree on requirements	Discussion with all stakeholders to gather a full set of requirements. Different personas must participate to agree on a subset of requirements to complete for each phase of the project.	One Week
Navigating the Microsoft Purview governance portal	Understand how to use Microsoft Purview from the home page.	One Day
Configure ADF for lineage	Identify key pipelines and data assets. Gather all information required to connect to an internal ADF account.	One Day
Scan a data source such as Azure Data Lake Storage Gen2 or a SQL server .	Add the data source and set up a scan. Ensure the scan successfully detects all assets.	Two Day
Search and browse	Allow end users to access Microsoft Purview and perform end-to-end search and browse scenarios.	One Day

Other helpful links

- [Create a Microsoft Purview account](#)
- [Create a collection](#)
- [Concept: Permissions and access](#)
- [Microsoft Purview product glossary](#)

Acceptance criteria

- Microsoft Purview account is created successfully in organization subscription under the organization tenant.
- A small group of users with multiple roles can access Microsoft Purview.
- Microsoft Purview is configured to scan at least one data source.
- Users should be able to extract key values of Microsoft Purview such as:
 - [Search and browse](#)
 - [Lineage](#)

- Users should be able to assign asset ownership in the asset page.
- Presentation and demo to raise awareness to key stakeholders.
- Buy-in from management to approve more resources for MVP phase.

Phase 2: Minimum viable product

Once you have the agreed requirements and participated business units to onboard Microsoft Purview, the next step is to work on a Minimum Viable Product (MVP) release. In this phase, you'll expand the usage of Microsoft Purview to more users who will have more needs horizontally and vertically. There will be key scenarios that must be met horizontally for all users such as glossary terms, search, and browse. There will also be in-depth requirements vertically for each business unit or group to cover specific end-to-end scenarios such as lineage from Azure Data Lake Storage to Azure Synapse DW to Power BI.

Tasks to complete

[\[\] Expand table](#)

Task	Detail	Duration
Scan Azure Synapse Analytics	Start to onboard your database sources and scan them to populate key assets	Two Days
Create custom classifications and rules	Once your assets are scanned, your users may realize that there are other use cases for more classification beside the default classifications from Microsoft Purview.	2-4 Weeks
Scan Power BI	If your organization uses Power BI, you can scan Power BI in order to gather all data assets being used by Data Scientists or Data Analysts that have requirements to include lineage from the storage layer.	1-2 Weeks
Import glossary terms	In most cases, your organization may already develop a collection of glossary terms and term assignment to assets. This will require an import process into Microsoft Purview via .csv file.	One Week
Add contacts to assets	For top assets, you may want to establish a process to either allow other personas to assign contacts or import via REST APIs.	One Week
Add sensitive labels and scan	This might be optional for some organizations, depending on the usage of Labeling from Microsoft 365.	1-2 Weeks

Task	Detail	Duration
Get classification and sensitive insights	For reporting and insight in Microsoft Purview, you can access this functionality to get various reports and provide presentation to management.	One Day
Onboard more users using Microsoft Purview managed users	This step will require the Microsoft Purview Admin to work with the Microsoft Entra Admin to establish new Security Groups to grant access to Microsoft Purview.	One Week

Other helpful links

- [Collections architecture best practices](#)
- [Classification best practices](#)
- [Glossary best practices](#)
- [Labeling best practices](#)
- [Asset lifecycle best practices](#)

Acceptance criteria

- Successfully onboard a larger group of users to Microsoft Purview (50+)
- Scan business critical data sources
- Import and assign all critical glossary terms
- Successfully test important labeling on key assets
- Successfully met minimum scenarios for participated business units' users

Phase 3: Pre-production

Once the MVP phase has passed, it's time to plan for pre-production milestone. You may want to include scanning on on-premises data sources such as SQL Server. If there's any gap in data sources not supported by Microsoft Purview, it's time to explore the Atlas API to understand other options.

Tasks to complete

 [Expand table](#)

Task	Detail	Duration
Refine your scan using scan rule set	Your organization will have many data sources for pre-production. It's important to pre-define key criteria for	1-2 Days

Task	Detail	Duration
	scanning so that classifications and file extension can be applied consistently across the board.	
Assess region availability for scan for each of your sources by checking source pages	Depending on the region of the data sources and organizational requirements on compliance and security, you may want to consider what regions must be available for scanning.	One Day
Understand firewall concept when scanning	This step requires some exploration of how the organization configures its firewall and how Microsoft Purview can authenticate itself to access the data sources for scanning.	One Day
Understand Private Link concept when scanning	If your organization uses Private Link, you must lay out the foundation of network security to include Private Link as a part of the requirements.	One Day
Scan on-premises SQL Server	This is optional if you have on-premises SQL Server. The scan will require setting up Self-hosted Integration Runtime and adding SQL Server as a data source.	1-2 Weeks
Use Microsoft Purview REST API for integration scenarios	If you have requirements to integrate Microsoft Purview with other third party technologies such as orchestration or ticketing system, you may want to explore REST API area.	1-4 Weeks
Understand Microsoft Purview pricing	This step will provide the organization important financial information to make decision.	1-5 Days

Other helpful links

- [Labeling best practices](#)
- [Network architecture best practices](#)

Acceptance criteria

- Successfully onboard at least one business unit with all of users
- Scan on-premises data source such as SQL Server
- POC at least one integration scenario using REST API
- Complete a plan to go to production, which should include key areas on infrastructure and security

Phase 4: Production

The above phases should be followed to create an effective data lifecycle management, which is the foundation for better governance programs. Data governance will help your organization prepare for the growing trends such as AI, Hadoop, IoT, and blockchain. It's just the start for many things data and analytics, and there's plenty more that can be discussed. The outcome of this solution would deliver:

- **Business Focused** - A solution that is aligned to business requirements and scenarios over technical requirements.
- **Future Ready** - A solution will maximize default features of the platform and use standardized industry practices for configuration or scripting activities to support the advancements/evolution of the platform.

Tasks to complete

expand table Expand table

Task	Detail	Duration
Scan production data sources with Firewall enabled	If this is optional when firewall is in place but it's important to explore options to hardening your infrastructure.	1-5 Days
Enable Private Link	If this is optional when Private Link is used. Otherwise, you can skip this as it's a must-have criterion when Private is enabled.	1-5 Days
Create automated workflow	Workflow is important to automate process such as approval, escalation, review and issue management.	2-3 Weeks
Create operation documentation	Data governance isn't a one-time project. It's an ongoing program to fuel data-driven decision making and creating opportunities for business. It's critical to document key procedure and business standards.	One Week

Other helpful links

- [Manage workflow runs](#)
- [Workflow requests and approvals](#)
- [Manage integration runtimes](#)
- [Request access to a data asset](#)

Acceptance criteria

- Successfully onboard all business unit and their users

- Successfully meet infrastructure and security requirements for production
- Successfully meet all use cases required by the users

Platform hardening

More hardening steps can be taken:

- Increase security posture by enabling scan on firewall resources or use Private Link
- Fine-tune scope scan to improve scan performance
- [Use REST APIs](#) to export critical metadata and properties for backup and recovery
- [Use workflow](#) to automate ticketing and eventing to avoid human errors
- [Use policies](#) to manage access to data assets through the Microsoft Purview governance portal.

Lifecycle considerations

Another important aspect to include in your production process is how classifications and labels can be migrated. Microsoft Purview has over 90 system classifiers. You can apply system or custom classifications on file, table, or column assets. Classifications are like subject tags and are used to mark and identify content of a specific type found within your data estate during scanning. Sensitivity labels are used to identify the categories of classification types within your organizational data, and then group the policies you wish to apply to each category. It makes use of the same sensitive information types as Microsoft 365, allowing you to stretch your existing security policies and protection across your entire content and data estate. It can scan and automatically classify documents. For example, if you have a file named multiple.docx and it has a National ID number in its content, Microsoft Purview will add classification such as EU National Identification Number in the Asset Detail page.

In the Microsoft Purview Data Map, there are several areas where the Catalog Administrators need to ensure consistency and maintenance best practices over its life cycle:

- **Data assets** – Data sources will need to be rescanned across environments. It's not recommended to scan only in development and then regenerate them using APIs in Production. The main reason is that the Microsoft Purview scanners do a lot more "wiring" behind the scenes on the data assets, which could be complex to move them to a different Microsoft Purview instance. It's much easier to just add the same data source in production and scan the sources again. The general best practice is to have documentation of all scans, connections, and authentication mechanisms being used.

- **Scan rule sets** – This is your collection of rules assigned to specific scan such as file type and classifications to detect. If you don't have that many scan rule sets, it's possible to just re-create them manually again via Production. This will require an internal process and good documentation. However, if your rule sets change on a daily or weekly basis, this could be addressed by exploring the REST API route.
- **Custom classifications** – Your classifications may not also change regularly. During the initial phase of deployment, it may take some time to understand various requirements to come up with custom classifications. However, once settled, this will require little change. So the recommendation here's to manually migrate any custom classifications over or use the REST API.
- **Glossary** – It's possible to export and import glossary terms via the UX. For automation scenarios, you can also use the REST API.
- **Resource set pattern policies** – This functionality is advanced for any typical organizations to apply. In some cases, your Azure Data Lake Storage has folder naming conventions and specific structure that may cause problems for Microsoft Purview to generate the resource set. Your business unit may also want to change the resource set construction with more customizations to fit the business needs. For this scenario, it's best to keep track of all changes via REST API, and document the changes through external versioning platform.
- **Role assignment** – This is where you control who has access to Microsoft Purview and which permissions they have. Microsoft Purview also has REST API to support export and import of users and roles but this isn't Atlas API-compatible. The recommendation is to assign an Azure Security Group and manage the group membership instead.

Moving tenants

Moving tenants is not currently supported for Microsoft Purview.

Moving subscriptions

It's possible to move your Microsoft Purview account between subscriptions. However, if your account was created before December 15, 2023 (or deployed using an API version previous to 2023-05-01-preview), or is using a managed Event Hubs, the managed storage account and managed Event Hubs associated with your Microsoft Purview account will not migrate with your instance. Your Microsoft Purview account will still be able to function, but you should not remove these resources.

If you need to remove the managed resources from the other subscription, you will need to create a new Microsoft Purview account and [migrate your information](#) to this

new account, before removing the original and its managed resources.

Next steps

- [Collections best practices](#)
 - [Navigate the home page and search for an asset](#)
-

Feedback

Was this page helpful?

 Yes

 No

Microsoft Purview glossary best practices

Article • 07/20/2023

The business glossary is a definition of terms specific to a domain of knowledge that is commonly used, communicated, and shared in organizations as they're conducting business. A common business glossary (for example, business language) is significant as it's critical in improving an organization's overall business productivity and performance. You observe in most organizations that their business language is being codified based on business dealings associated with:

- Business Meetings
- Stand-Ups, Projects, and Systems (ERP, CRM, SharePoint, etc.).
- Business Plans and Business Processes
- Presentations
- Reporting and Business Rules
- Learnings (Knowledge Acquisition)
- Business Models
- Policy and Procedure

It's important that your organization's business language and discourse align to a common business glossary to ensure your organization's data assets are properly applied in conducting business at speed and with agility as rapid changing business needs happen.

This article is intended to provide guidance around how to establish and govern your organization's A to Z business glossary of commonly used terms and it's aimed to provide more guidance to ensure you're able to focus on promoting a shared understanding for business data governance ownership. Adopting these considerations and recommendations will help your organization achieve success with Microsoft Purview. The adoption by your organization of the business glossary will depend on you promoting consistent use of a business glossary to enable your organization to better understand the meanings of their business terms they come across daily while running the organization's business.

Why is a common business glossary needed?

Without a common business glossary, an organization's performance, culture, operations, and strategy often will adversely hinder the business. You'll observe, in this hindrance, a condition in which cultural differences arise grounded in an inconsistent

business language. These inconsistencies about the business language are communicated between team members and prevents them from using their relevant data assets as a competitive advantage. You'll also observe when there are language barriers, in which, most organizations will spend more time pursuing nonproductive and noncollaborative activities as they need to rely on more detailed interactions to reach the same meaning and understanding for their data assets.

Recommendations for implementing new glossary terms

Creating terms is necessary to build the business vocabulary and apply it to assets within Microsoft Purview. When a new Microsoft Purview account is created, by default, there are no built-in terms in the account.

This creation process should follow strict naming standards to ensure that the glossary doesn't contain duplicate or competing terms.

- Establish a strict hierarchy for all business terms across your organization.
- The hierarchy could consist of a business domain such as: Finance, Marketing, Sales, HR, etc.
- Implement naming standards for all glossary terms that include case sensitivity. In Microsoft Purview terms are case-sensitive.
- Always use the provide search glossary terms feature before adding a new term. This will help you avoid adding duplicate terms to the glossary.
- Avoid deploying terms with duplicated names. In Microsoft Purview, terms with the same name can exist under different parent terms. This can lead to confusion and should be well thought out before building your business glossary to avoid duplicated terms.

Glossary terms in Microsoft Purview are case sensitive and allow white space. The following shows a poorly executed example of implementing glossary terms and demonstrates the confusion caused:

Glossary terms

[+ New term](#)[Manage term templates](#)[Edit](#) account number[Term t](#)

Showing 7 terms

**Name****Address****Account Number****customer name****Sales****Account Number****Account number****Account Number**

As a best practice it always best to: Plan, search, and strictly follow standards. The following shows an approach that is better thought out and greatly reduces confusion between glossary terms:

Glossary terms



New term



Manage term templates



account number



Showing 5 terms



Name



Customer



Customer **Account Number**



Customer name



Vendor



Vendor **Account Number**

Recommendations for deploying glossary term templates

When building new term templates in Microsoft Purview, review the following considerations:

- Term templates are used to add custom attributes to glossary terms.
- By default, Microsoft Purview offers several [out-of-the-box term attributes](#) such as Name, Nick Name, Status, Definition, Acronym, Resources, Related terms,

Synonyms, Stewards, Experts, and Parent term, which are found in the "System Default" template.

- Default attributes can't be edited or deleted.
- Custom attributes extend beyond default attributes, allowing the data curators to add more descriptive details to each term to completely describe the term in the organization.
- As a reminder, Microsoft Purview stores only meta-data. Attributes should describe the meta-data; not the data itself.
- Keep definition simple. If there are custom metrics or formulas these could easily be added as an attribute.
- A term must include at least default attributes. When building new glossary terms if you use custom templates, other attributes that are included in the custom template are expected for the given term.

Recommendations for importing glossary terms from term templates

- Terms may be imported with the "System default" or custom templates.
- When importing terms, use the sample .CSV file to guide you. This can save hours of frustration.
- When importing terms from a .CSV file, be sure that terms already existing in Microsoft Purview are intended to be updated. When using the import feature, Microsoft Purview will overwrite existing terms.
- Before importing terms, test the import in a lab environment to ensure that no unexpected results occur, such as duplicate terms.
- The email address for Stewards and Experts should be the primary address of the user from the Microsoft Entra group. Alternate email, user principal name and non-Microsoft Entra ID emails aren't yet supported.
- Glossary terms provide four status: Draft, Approved, Expired, Alert. Draft isn't officially implemented, Approved is official/stand/approved for production, Expired means should no longer be used, Alert need to pay more attention. For more information, see [the import and export glossary terms article](#).

Recommendations for exporting glossary terms

Exporting terms may be useful in Microsoft Purview account to account, backup, or disaster recovery scenarios.

Recommendations for multi-glossary

Microsoft Purview supports having multiple business glossaries managed within Microsoft Purview. Consider using multiple glossaries when:

- The same term has different meanings across regions/departments/organizations/teams/etc.
- You want to give management of the glossary to experts in their separate regions/departments/organizations/teams/etc.
- Glossary terms and needs are different and have no overlap between regions/departments/organizations/teams/etc.

Glossary Management

- Related terms are bi-directional. For example: If I have term A and term B. I add B as a related term to term A. Now term B appears under term A's related terms, and term A appears under term B's related terms.

Recommendations for assigning terms to assets

- While classifications and sensitivity labels are applied to assets automatically by the system based on classification rules, glossary terms aren't applied automatically.
- Similar to classifications, glossary terms can be mapped to assets at the asset level or scheme level.
- In Microsoft Purview, terms can be added to assets in different ways:
 - Manually, using the Microsoft Purview governance portal.
 - Using Bulk Edit mode to update up to 25 assets, using the Microsoft Purview governance portal.
 - Curated Code using the Atlas API.
- Use Bulk Edit Mode when assigning terms manually. This feature allows a curator to assign glossary terms, owners, experts, classifications and certified in bulk based on selected items from a search result. Multiple searches can be chained by selecting objects in the results. The Bulk Edit will apply to all selected objects. Be sure to clear the selections after the bulk edit has been performed.
- Other bulk edit operations can be performed by using the Atlas API. An example would be using the API to add descriptions or other custom properties to assets in bulk programmatically.

Next steps

- Import and export glossary terms

- Best practices for describing data with terms, tags, managed attributes, and business assets
- Create and manage glossaries
- Create and manage terms
- Manage term templates
- Browse the data catalog in Microsoft Purview

Labeling best practices for the data map

Article • 07/20/2023

ⓘ Note

These best practices cover applying labels to the data map in [Microsoft Purview unified governance solutions](#). For more information about labeling in Microsoft Purview risk and compliance solutions, [go here](#). For more information about Microsoft Purview in general, [go here](#).

The Microsoft Purview Data Map supports labeling structured and unstructured data stored across various data sources. Labeling data within the data map allows users to easily find data that matches predefined autolabeling rules that were configured in the Microsoft Purview compliance portal. The data map extends the use of sensitivity labels from Microsoft Purview Information Protection to assets stored in infrastructure cloud locations and structured data sources.

Protect personal data with custom sensitivity labels for Microsoft Purview

Storing and processing personal data is subject to special protection. Labeling personal data is crucial to help you identify sensitive information. You can use the detection and labeling tasks for personal data in different stages of your workflows. Because personal data is ubiquitous and fluid in your organization, you need to define identification rules for building policies that suit your individual situation.

Why do you need to use labeling within the data map?

With the data map, you can extend your organization's investment in sensitivity labels from Microsoft Purview Information Protection to assets that are stored in files and database columns within Azure, multicloud, and on-premises locations. These locations are defined in [supported data sources](#). When you apply sensitivity labels to your content, you can keep your data secure by stating how sensitive certain data is in your organization. The data map also abstracts the data itself, so you can use labels to track the type of data, without exposing sensitive data on another platform.

Microsoft Purview Data Map labeling best practices and considerations

The following sections walk you through the process of implementing labeling for your assets.

Get started

- To enable sensitivity labeling in the data map, follow the steps in [automatically apply sensitivity labels to your data in the Microsoft Purview Data Map](#).
- To find information on required licensing and helpful answers to other questions, see [Sensitivity labels in the Microsoft Purview Data Map FAQ](#).

Label considerations

- If you already have sensitivity labels from Microsoft Purview Information Protection in use in your environment, continue to use your existing labels. Don't make duplicate or more labels for the data map. This approach allows you to maximize the investment you've already made in the Microsoft Purview. It also ensures consistent labeling across your data estate.
- If you haven't created sensitivity labels in Microsoft Purview Information Protection, review the documentation to [get started with sensitivity labels](#). Creating a classification schema is a tenant-wide operation. Discuss it thoroughly before you enable it within your organization.

Label recommendations

- When you configure sensitivity labels for the Microsoft Purview Data Map, you might define autolabeling rules for files, database columns, or both within the label properties. Microsoft Purview labels files within the Microsoft Purview Data Map. When the autolabeling rule is configured, Microsoft Purview automatically applies the label or recommends that the label is applied.

Warning

If you haven't configured autolabeling for items on your sensitivity labels, users might be affected within your Office and Microsoft 365 environment. You can test autolabeling on database columns without affecting users.

- If you're defining new autolabeling rules for files when you configure labels for the Microsoft Purview Data Map, make sure that you have the condition for applying the label set appropriately.
- You can set the detection criteria to **All of these** or **Any of these** in the upper right of the autolabeling for items page of the label properties.
- The default setting for detection criteria is **All of these**. This setting means that the asset must contain all the specified sensitive information types for the label to be applied. While the default setting might be valid in some instances, many customers want to use **Any of these**. Then if at least one asset is found, the label is applied.

(!) Note

Trainable classifiers from Microsoft Purview Information Protection aren't supported by Microsoft Purview Data Map.

- Maintain consistency in labeling across your data estate. If you use autolabeling rules for files, use the same sensitive information types for autolabeling database columns.
- [Define your sensitivity labels via Microsoft Purview Information Protection to identify your personal data at a central place.](#)
- [Use policy templates as a starting point to build your rule sets.](#)
- [Combine data classifications to an individual rule set.](#)
- [Force labeling by using autolabel functionality.](#)
- Build groups of sensitivity labels and store them as a dedicated sensitivity label policy. For example, store all required sensitivity labels for regulatory rules by using

the same sensitivity label policy to publish.

- Capture all test cases for your labels. Test your label policies with all applications you want to secure.
- Promote sensitivity label policies to the Microsoft Purview Data Map.
- Run test scans from the Microsoft Purview Data Map on different data sources like hybrid cloud and on-premises to identify sensitivity labels.
- Gather and consider insights, for example, by using Microsoft Purview Data Estate Insights. Use alerting mechanisms to mitigate potential breaches of regulations.

By using sensitivity labels with Microsoft Purview Data Map, you can extend information protection beyond the border of your Microsoft data estate to your on-premises, hybrid cloud, multicloud, and software as a service (SaaS) scenarios.

Next steps

- [Get started with sensitivity labels.](#)
- [How to automatically apply sensitivity labels to your data in the Microsoft Purview Data Map.](#)

Feedback

Was this page helpful?

 Yes

 No

Microsoft Purview Data Lineage best practices

Article • 01/30/2024

Data Lineage is broadly understood as the lifecycle that spans the data's origin, and where it moves over time across the data estate. Microsoft Purview can capture lineage for data in different parts of your organization's data estate, and at different levels of preparation including:

- Raw data staged from various platforms
- Transformed and prepared data
- Data used by visualization platforms

Why do you need adopt Lineage?

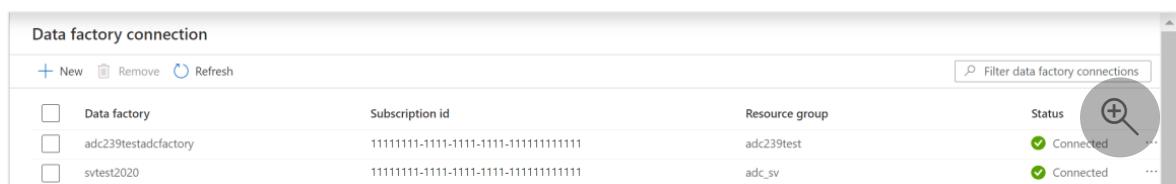
Data lineage is the process of describing what data exists, where it's stored and how it flows between systems. There are many reasons why data lineage is important, but at a high level these can all be boiled down to three categories that we'll explore here:

- Track data in reports
- Impact analysis
- Capture the changes and where the data has resided through the data life cycle

Azure Data Factory Lineage best practice and considerations

Azure Data Factory instance

- Data lineage won't be reported to the catalog automatically until the Data Factory connection status turns to Connected. The rest of status Disconnected and CannotAccess can't capture lineage.



Data factory connection	Subscription id	Resource group	Status
<input type="checkbox"/> Data factory	11111111-1111-1111-1111-111111111111	adc239test	Connected
<input type="checkbox"/> adc239testadcfactory	11111111-1111-1111-1111-111111111111	adc_sv	Connected
<input type="checkbox"/> svtest2020			

- Each Data Factory instance can connect to only one Microsoft Purview account. You can establish new connection in another Microsoft Purview account, but this

will turn existing connection to disconnected.

New Data Factory connections

Each Data Factory account can connect to only one Purview account.

Azure Subscription

▼

Data Factory *

▼

1 selected

⚠ A new connection will be established between current Purview account and selected Data Factory accounts. By clicking 'OK' any Data Factory connected to another Purview account specified under 'Existing connection' will be disconnected.

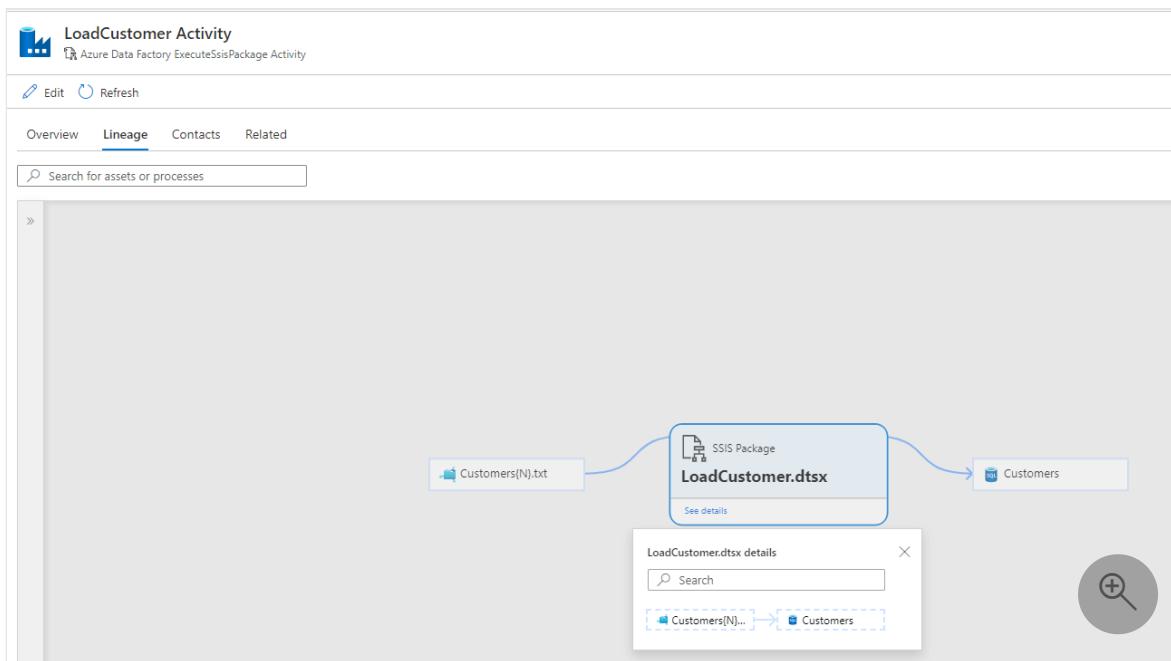
Data Factory	Existing connection
E2ETest-Lineage	prdu-05082020 

- Data factory's managed identity is used to authenticate lineage push operations in Microsoft Purview account. The data factory's managed identity needs Data Curator role on Microsoft Purview root collection.
- Currently, only 10 data factories can be connected at a time. If you want to add more than 10 data factories, please add 10 new data factory connections at a time using the wizard or use API to connect more than 10 data factories in one operation.

Azure Data Factory activities

- Microsoft Purview captures runtime lineage from the following Azure Data Factory activities:
 - [Copy activity](#)
 - [Data Flow activity](#)
 - [Execute SSIS Package activity](#)
- Microsoft Purview drops lineage if the source or destination uses an unsupported data storage system.
 - Supported data sources in copy activity are listed [Copy activity support of Connect to Azure Data Factory](#)
 - Supported data sources in data flow activity are listed [Data Flow support of Connect to Azure Data Factory](#)

- Supported data sources in SSIS are listed [SSIS execute package activity support of Lineage from SQL Server Integration Services](#)
- Microsoft Purview can't capture lineage if Azure Data Factory copy activity uses copy activity features listed in [Limitations on copy activity lineage of Connect to Azure Data Factory](#)
- For the lineage of Dataflow activity, Microsoft Purview only support source and sink. The lineage for Dataflow transformation isn't supported yet.
- Data flow lineage doesn't integrate with Microsoft Purview resource set. **Resource set example:**
Qualified name: <https://myblob.blob.core.windows.net/sample-data/data{N}.csv>
Display name: "data"
- For the lineage of Execute SSIS Package activity, we only support source and destination. The lineage for transformation isn't supported yet.



- Please refer the following step-by-step guide to [push Azure Data Factory lineage in Microsoft Purview](#).

Build custom lineage manually or with REST APIs

One of the important platform features of Microsoft Purview is the ability to show the lineage between datasets created by data processes. Systems like Data Factory, Data Share, and Power BI capture the lineage of data as it moves. In certain situations, automatically generated lineage by Purview is incomplete or missing for practical

visualization and/or enterprise reporting purposes. In those scenarios, you can create custom lineage entries manually in the Microsoft Purview portal, or via Apache Atlas hooks and the REST API. Another major benefit of using REST APIs to report or build custom lineage is to overcome or mitigate the limitations of functionality exposed by Manual Lineage.

To build custom lineage manually, you can follow this user guide: [Manual lineage entries in Microsoft Purview](#).

To build custom lineage in Microsoft Purview using the REST APIs, follow this user guide: [Microsoft Purview - Building Custom Lineage using REST APIs](#).

💡 Tip

In some cases, the REST APIs can provide more input and customization options than building the lineage entries manually through the portal.

Next steps

- [Manage data sources](#)

Feedback

Was this page helpful?

 Yes

 No

Microsoft Purview network architecture and best practices

Article • 11/13/2023

Microsoft Purview governance solutions are a platform as a service (PaaS) solutions for data governance. Microsoft Purview accounts have public endpoints that are accessible through the internet to connect to the service. However, all endpoints are secured through Microsoft Entra logins and role-based access control (RBAC).

Note

These best practices cover the network architecture for **Microsoft Purview unified governance solutions**. For more information about Microsoft Purview risk and compliance solutions, [go here](#). For more information about Microsoft Purview in general, [go here](#).

For an added layer of security, you can create private endpoints for your Microsoft Purview account. You'll get a private IP address from your virtual network in Azure to the Microsoft Purview account and its managed resources. This address will restrict all traffic between your virtual network and the Microsoft Purview account to a private link for user interaction with the APIs and Microsoft Purview governance portal, or for scanning and ingestion.

Currently, the Microsoft Purview firewall provides access control for the public endpoint of your purview account. You can use the firewall to allow all access or to block all access through the public endpoint when using private endpoints. For more information see, [Microsoft Purview firewall options](#)

Based on your network, connectivity, and security requirements, you can set up and maintain Microsoft Purview accounts to access underlying services or ingestion. Use this best practices guide to define and prepare your network environment so you can access Microsoft Purview and scan data sources from your network or cloud.

This guide covers the following network options:

- Use [Azure public endpoints](#).
- Use [private endpoints](#).
- Use [private endpoints and allow public access on the same Microsoft Purview account](#).

- Use Azure [public endpoints](#) to access Microsoft Purview governance portal and [private endpoints](#) for ingestion.

This guide describes a few of the most common network architecture scenarios for Microsoft Purview. Though you're not limited to those scenarios, keep in mind the [limitations](#) of the service when you're planning networking for your Microsoft Purview accounts.

Prerequisites

To understand which network option is best for your environment, we suggest that you perform the following actions first:

- Review your network topology and security requirements before registering and scanning any data sources in Microsoft Purview. For more information, see: [Define an Azure network topology](#).
- Define your [network connectivity model for PaaS services](#).

Option 1: Use public endpoints

By default, you can use Microsoft Purview accounts through the public endpoints accessible over the internet. Allow public networks in your Microsoft Purview account if you have the following requirements:

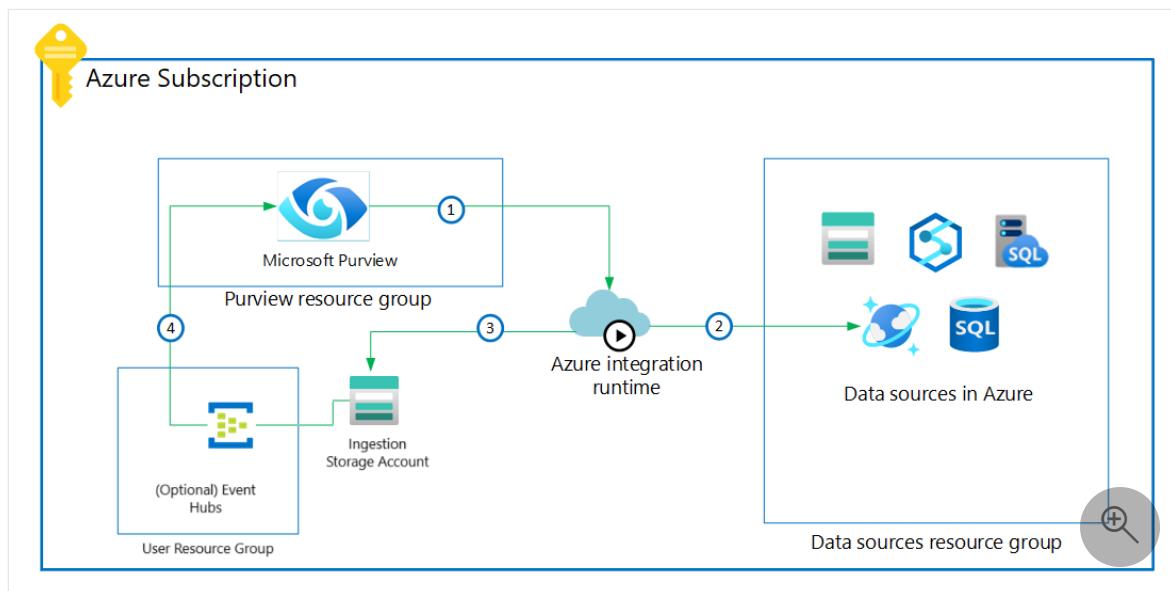
- No private connectivity is required when scanning or connecting to Microsoft Purview endpoints.
- All data sources are software-as-a-service (SaaS) applications only.
- All data sources have a public endpoint that's accessible through the internet.
- Business users require access to a Microsoft Purview account and the Microsoft Purview governance portal through the internet.

Integration runtime options

To scan data sources while the Microsoft Purview account firewall is set to allow public access, you can use all the integration runtime types - the Azure integration runtime, [Managed VNet integration runtime](#), and a [self-hosted integration runtime](#). Learn more from [Choose the right integration runtime configuration for your scenario](#).

Here are some best practices:

- Whenever applicable, we recommend that you use the Azure integration runtime or Managed VNet integration runtime to scan data sources, to reduce cost and administrative overhead.
- The following steps show the communication flow at a high level when you're using the Azure integration runtime to scan a data source:

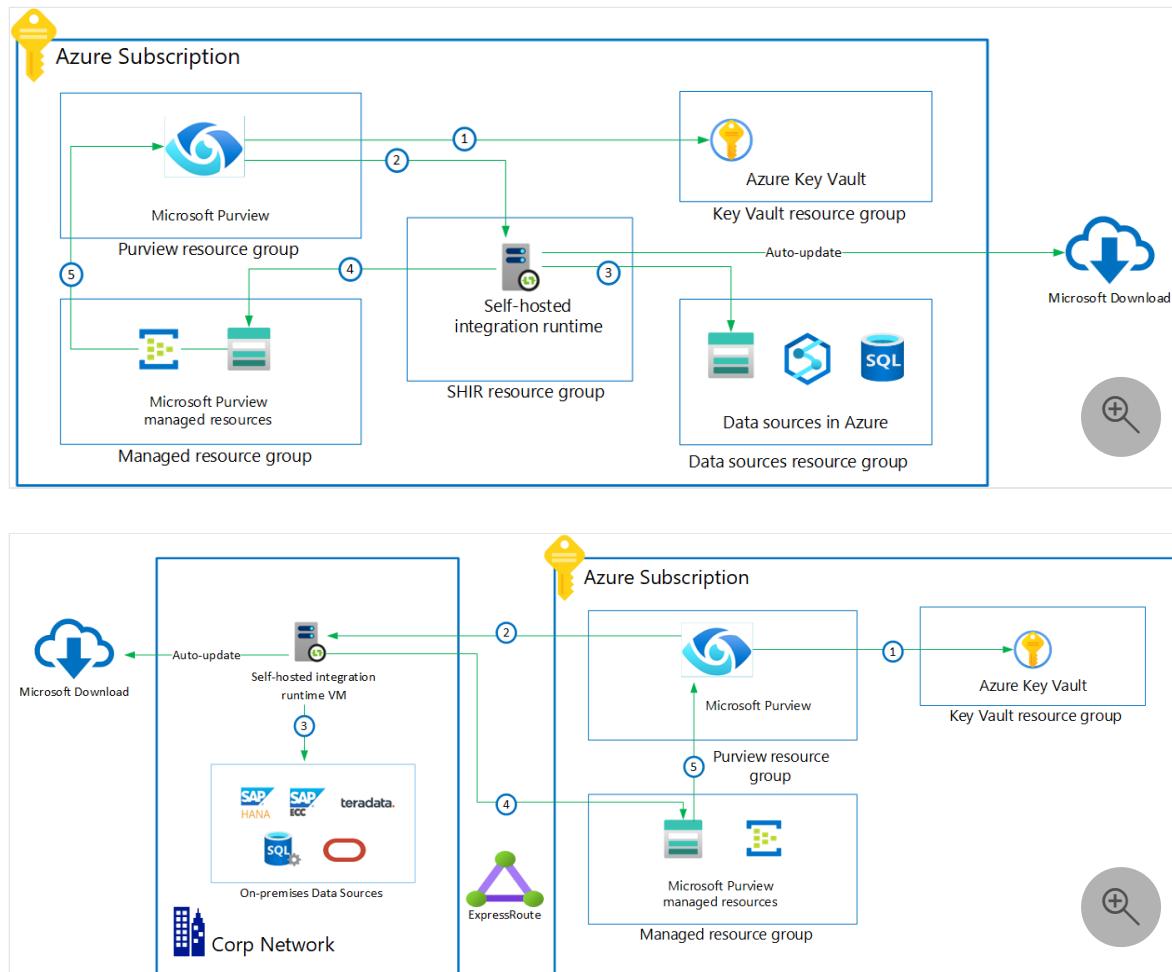


① Note

This graphic only applies to Microsoft Purview accounts created after December 15, 2023 (or deployed using API version 2023-05-01-preview onwards).

1. A manual or automatic scan is initiated from the Microsoft Purview Data Map through the Azure integration runtime.
 2. The Azure integration runtime connects to the data source to extract metadata.
 3. Metadata is queued in the Microsoft Purview ingestion storage account and stored in Azure Blob Storage temporarily.
 4. Metadata is sent to the Microsoft Purview Data Map.
- Scanning on-premises and VM-based data sources always requires using a self-hosted integration runtime. The Azure integration runtime isn't supported for these data sources. The following steps show the communication flow at a high level when you're using a self-hosted integration runtime to scan a data source. The first diagram shows a scenario where resources are within Azure or on a VM in

Azure. The second diagram shows a scenario with on-premises resources. The steps between the two are the same from Microsoft Purview's perspective:



1. A manual or automatic scan is triggered. Microsoft Purview connects to Azure Key Vault to retrieve the credential to access a data source.
2. The scan is initiated from the Microsoft Purview Data Map through a self-hosted integration runtime.
3. The self-hosted integration runtime service from the VM or on-premises machine connects to the data source to extract metadata.
4. Metadata is processed in the machine's memory for the self-hosted integration runtime. Metadata is queued in Microsoft Purview ingestion storage and then stored in Azure Blob Storage temporarily. Actual data never leaves the boundary of your network.
5. Metadata is sent to the Microsoft Purview Data Map.

Authentication options

When you're scanning a data source in Microsoft Purview, you need to provide a credential. Microsoft Purview can then read the metadata of the assets from the data source by using the integration runtime. Refer to each [data source article](#) for details about the supported authentication types and the needed permissions. Authentication options and requirements vary based on the following factors:

- **Data source type.** For example, if the data source is Azure SQL Database, you need to use a login with db_datareader access to each database. This can be a user-assigned managed identity or a Microsoft Purview managed identity. Or it can be a service principal in Microsoft Entra ID added to SQL Database as db_datareader.

If the data source is Azure Blob Storage, you can use a Microsoft Purview managed identity, or a service principal in Microsoft Entra ID added as a Blob Storage Data Reader role on the Azure storage account. Or use the storage account's key.

- **Authentication type.** We recommend that you use a Microsoft Purview managed identity to scan Azure data sources when possible, to reduce administrative overhead. For any other authentication types, you need to [set up credentials for source authentication inside Microsoft Purview](#):

1. Generate a secret inside an Azure key vault.
2. Register the key vault inside Microsoft Purview.
3. Inside Microsoft Purview, create a new credential by using the secret saved in the key vault.

- **Runtime type that's used in the scan.** Currently, you can't use a Microsoft Purview managed identity with a self-hosted integration runtime.

Other considerations

- If you choose to scan data sources using public endpoints, your self-hosted integration runtime VMs must have outbound access to data sources and Azure endpoints.
- Your self-hosted integration runtime VMs must have [outbound connectivity to Azure endpoints](#).

Option 2: Use private endpoints

Similar to other PaaS solutions, Microsoft Purview doesn't support deploying directly into a virtual network. So you can't use certain networking features with the offering's resources, such as network security groups, route tables, or other network-dependent appliances such as Azure Firewall. Instead, you can use private endpoints that can be

enabled on your virtual network. You can then disable public internet access to securely connect to Microsoft Purview.

You must use private endpoints for your Microsoft Purview account if you have any of the following requirements:

- You need to have end-to-end network isolation for Microsoft Purview accounts and data sources.
- You need to [block public access](#) to your Microsoft Purview accounts.
- Your platform-as-a-service (PaaS) data sources are deployed with private endpoints, and you've blocked all access through the public endpoint.
- Your on-premises or infrastructure-as-a-service (IaaS) data sources can't reach public endpoints.

Design considerations

- To connect to your Microsoft Purview account privately and securely, you need to deploy an account and a portal private endpoint. For example, this deployment is necessary if you intend to connect to Microsoft Purview through the API or use the Microsoft Purview governance portal.
- If you need to connect to the Microsoft Purview governance portal by using private endpoints, you have to deploy both account and portal private endpoints.
- To scan data sources through private connectivity, you need to configure at least one account and one ingestion private endpoint for Microsoft Purview.
- Review [DNS requirements](#). If you're using a custom DNS server on your network, clients must be able to resolve the fully qualified domain name (FQDN) for the Microsoft Purview account endpoints to the private endpoint's IP address.

Integration runtime options

To scan data sources through private connectivity, you can use [Managed VNet integration runtime](#) or [self-hosted integration runtime](#). Learn more from [Choose the right integration runtime configuration for your scenario](#).

- Whenever applicable, we recommend that you use the Managed VNet integration runtime to scan data sources, to reduce cost and administrative overhead.
- If using self-hosted integration runtime, you need to set up and use a self-hosted integration runtime on a Windows virtual machine that's deployed inside the same

or a peered virtual network where Microsoft Purview ingestion private endpoints are deployed.

- To scan on-premises data sources, you can also install a self-hosted integration runtime either on an on-premises Windows machine or on a VM inside an Azure virtual network.
- When you're using private endpoints with Microsoft Purview, you need to allow network connectivity from data sources to the self-hosted integration VM on the Azure virtual network where Microsoft Purview private endpoints are deployed.
- We recommend allowing automatic upgrade of the self-hosted integration runtime. Make sure you open required outbound rules in your Azure virtual network or on your corporate firewall to allow automatic upgrade. For more information, see [Self-hosted integration runtime networking requirements](#).

Authentication options

- Make sure that your credentials are stored in an Azure key vault and registered inside Microsoft Purview.
- You must create a credential in Microsoft Purview based on each secret that you create in the Azure key vault. You need to assign, at minimum, *get* and *list* access for secrets for Microsoft Purview on the Key Vault resource in Azure. Otherwise, the credentials won't work in the Microsoft Purview account.

Current limitations

- Scanning multiple Azure sources by using the entire subscription or resource group through ingestion private endpoints and a self-hosted integration runtime or Managed VNet integration runtime isn't supported when you're using private endpoints for ingestion. Instead, you can register and scan data sources individually.
- For limitations related to Microsoft Purview private endpoints, see [Known limitations](#).
- For limitations related to the Private Link service, see [Azure Private Link limits](#).

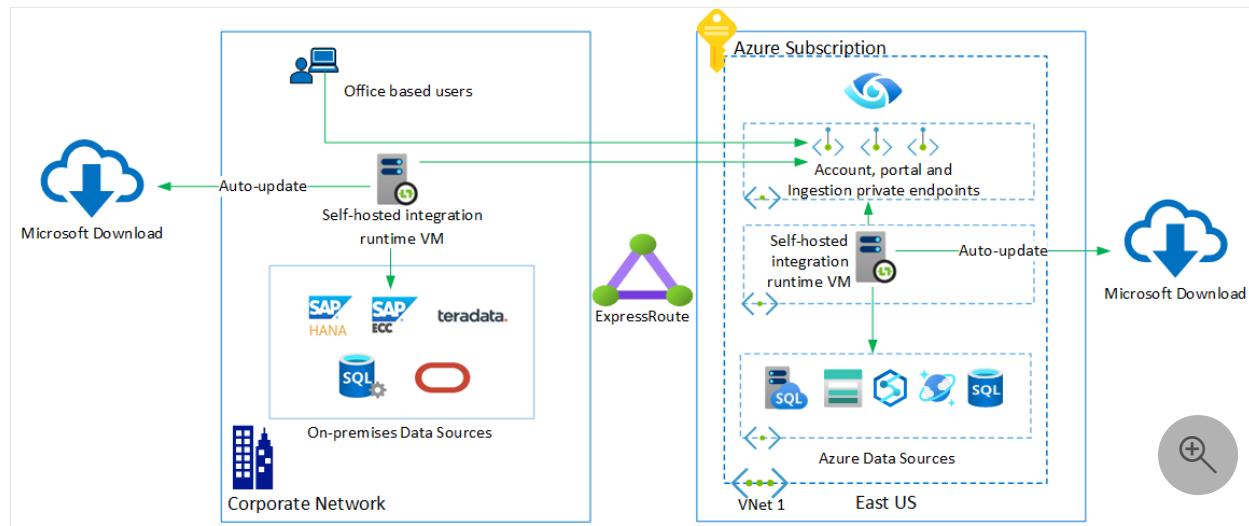
Private endpoint scenarios

Single virtual network, single region

In this scenario, all Azure data sources, self-hosted integration runtime VMs, and Microsoft Purview private endpoints are deployed in the same virtual network in an Azure subscription.

If on-premises data sources exist, connectivity is provided through a site-to-site VPN or Azure ExpressRoute connectivity to an Azure virtual network where Microsoft Purview private endpoints are deployed.

This architecture is suitable mainly for small organizations or for development, testing, and proof-of-concept scenarios.



Single region, multiple virtual networks

To connect two or more virtual networks in Azure together, you can use [virtual network peering](#). Network traffic between peered virtual networks is private and is kept on the Azure backbone network.

Many customers build their network infrastructure in Azure by using the hub-and-spoke network architecture, where:

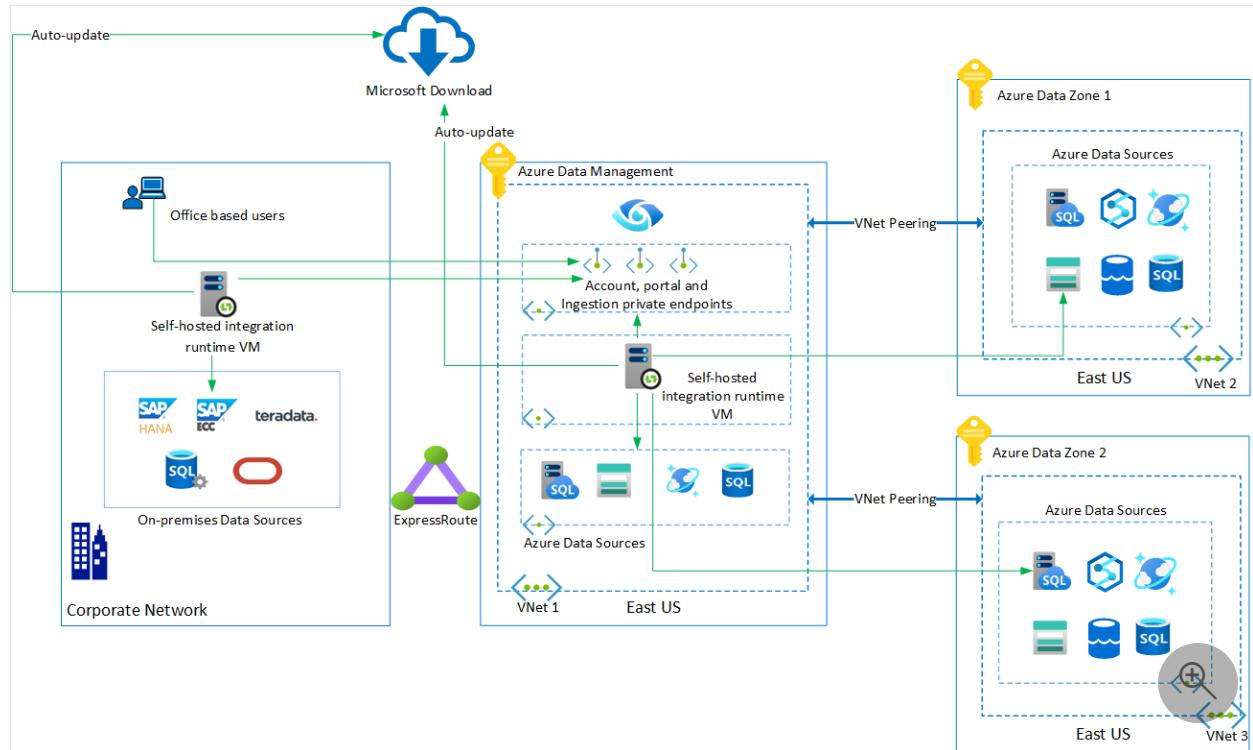
- Networking shared services (such as network virtual appliances, ExpressRoute/VPN gateways, or DNS servers) are deployed in the hub virtual network.
- Spoke virtual networks consume those shared services via virtual network peering.

In hub-and-spoke network architectures, your organization's data governance team can be provided with an Azure subscription that includes a virtual network (hub). All data services can be located in a few other subscriptions connected to the hub virtual network through a virtual network peering or a site-to-site VPN connection.

In a hub-and-spoke architecture, you can deploy Microsoft Purview and one or more self-hosted integration runtime VMs in the hub subscription and virtual network. You

can register and scan data sources from other virtual networks from multiple subscriptions in the same region.

The self-hosted integration runtime VMs can be deployed inside the same Azure virtual network or a peered virtual network where the account and ingestion private endpoints are deployed.

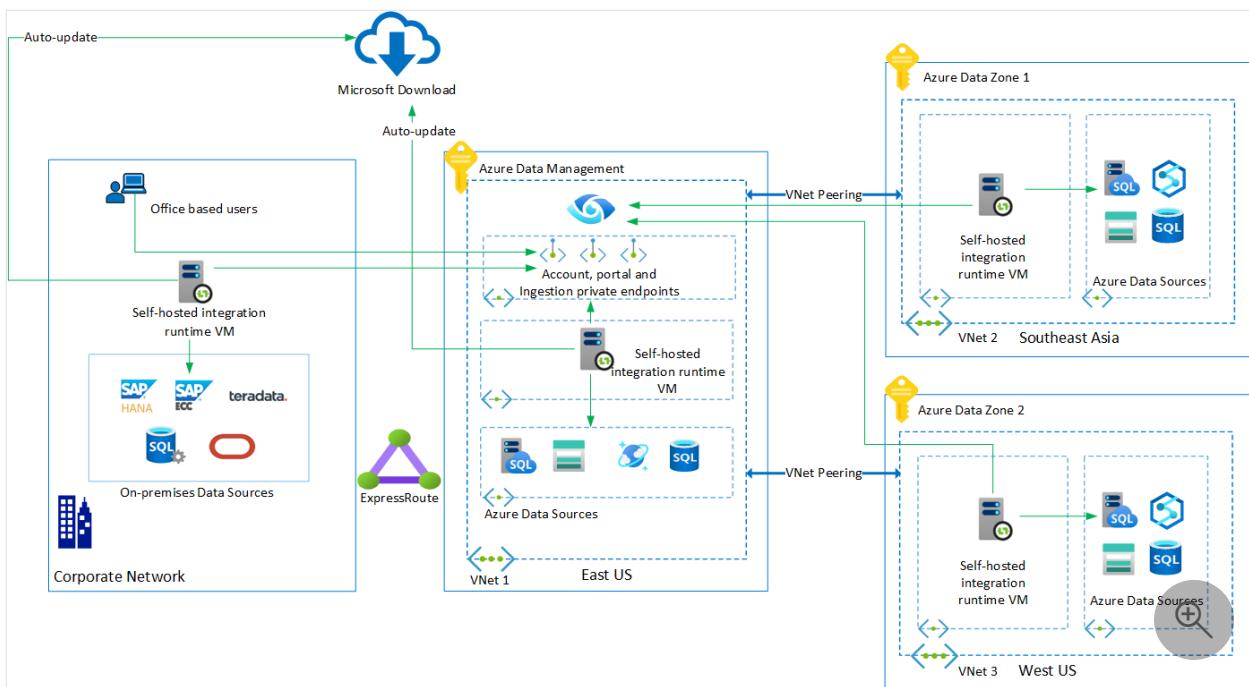


You can optionally deploy another self-hosted integration runtime in the spoke virtual networks.

Multiple regions, multiple virtual networks

If your data sources are distributed across multiple Azure regions in one or more Azure subscriptions, you can use this scenario.

For performance and cost optimization, we highly recommended deploying one or more self-hosted integration runtime VMs in each region where data sources are located.

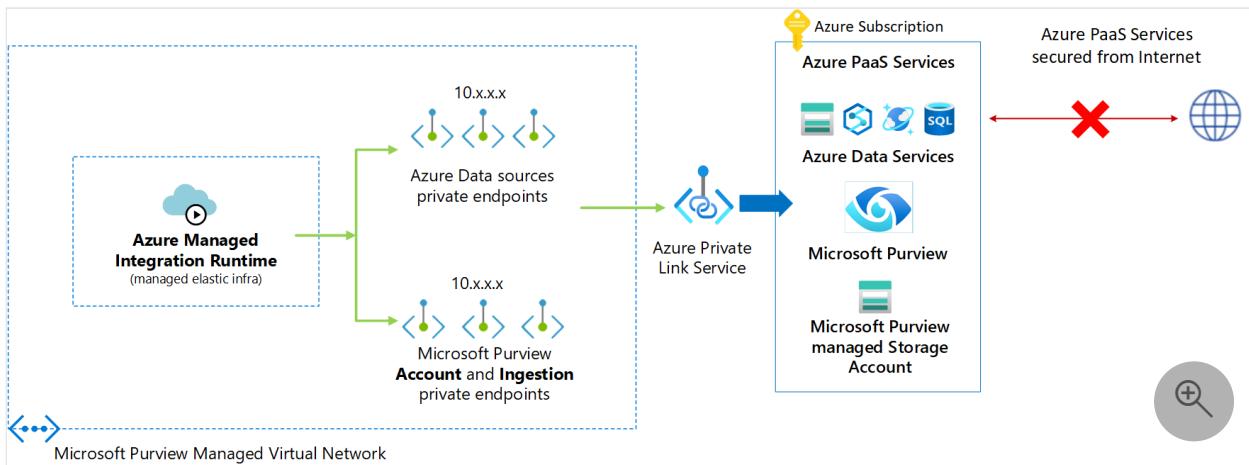


Scan using Managed Vnet Runtime

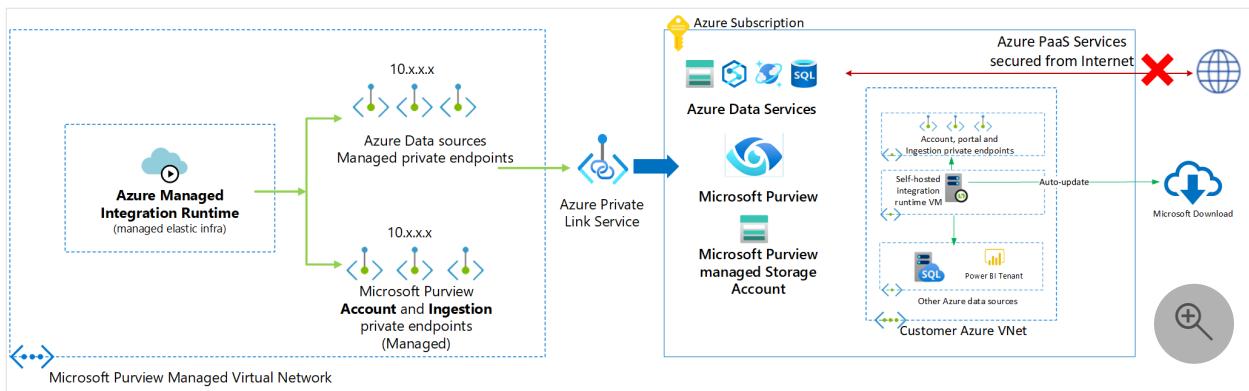
You can use Managed VNet Runtime to scan data sources in a private network. Learn more from [Use a Managed VNet with your Microsoft Purview account](#).

Using Managed VNet Runtime helps to minimize the administrative overhead of managing the runtime and reduce overall scan duration.

To scan any Azure data sources over private network using Managed VNet Runtime, a managed private endpoint must be deployed within Microsoft Purview Managed Virtual Network, even if the data source already has a private network in your Azure subscription.



If you need to scan on-premises data sources or additional data sources in Azure that are not supported by Managed VNet Runtime, you can deploy both Managed VNet Runtime and Self-hosted integration runtime.



If Microsoft Purview isn't available in your primary region

Microsoft Purview is an Azure platform as a service solution. You can deploy a Microsoft Purview account inside your Azure subscription in [any supported Azure regions](#).

If Microsoft Purview isn't available in your primary Azure region, consider the following factors when choosing a secondary region to deploy your Microsoft Purview account:

- Review the latency between your primary Azure region where data sources are deployed and your secondary Azure region, where Microsoft Purview account will be deployed. For more information, see [Azure network round-trip latency statistics](#).
- Review your data residency requirements. When you scan data sources in the Microsoft Purview Data Map, information related to your metadata is ingested and stored inside your data map in the Azure region where your Microsoft Purview account is deployed. For more information, see [Where is metadata stored](#).
- Review your network and security requirements if private network connectivity for user access or metadata ingestion is required. For more information, see [If Microsoft Purview isn't available in your primary region](#).

Option 1: Deploy your Microsoft Purview account in a secondary region and deploy all private endpoints in the primary region, where your Azure data sources are located. For this scenario:

- This is the recommended option, if Australia Southeast is the primary region for all your data sources and you have all network resources deployed in your primary region.
- Deploy a Microsoft Purview account in your secondary region (for example, Australia East).
- Deploy all Microsoft Purview private endpoints including account, portal and ingestion in your primary region (for example, Australia Southeast).
- Deploy all [Microsoft Purview self-hosted integration runtime](#) VMs in your primary region (for example, Australia Southeast). This helps to reduce cross region traffic as the Data Map scans will happen in the local region where data sources are

located and only metadata is ingested into your secondary region where your Microsoft Purview account is deployed.

- If you use [Microsoft Purview Managed VNets](#) for metadata ingestion, Managed VNet Runtime and all managed private endpoints will be automatically deployed in the region where your Microsoft Purview is deployed (for example, Australia East).

Option 2: Deploy your Microsoft Purview account in a secondary region and deploy private endpoints in the primary and secondary regions. For this scenario:

- This option is recommended if you have data sources in both primary and secondary regions and users are connected through the primary region.
- Deploy a Microsoft Purview account in your secondary region (for example, Australia East).
- Deploy Microsoft Purview governance portal private endpoint in the primary region (for example, Australia Southeast) for user access to Microsoft Purview governance portal.
- Deploy Microsoft Purview account and ingestion private endpoints in your primary region (for example, Australia southeast) to scan data sources locally in the primary region.
- Deploy Microsoft Purview account and ingestion private endpoints in your secondary region (for example, Australia East) to scan data sources locally in the secondary region.
- Deploy [Microsoft Purview self-hosted integration runtime](#) VMs in both primary and secondary regions. This will help to keep data Map scan traffic in the local region and send only metadata to Microsoft Purview Data Map where is configured in your secondary region (for example, Australia East).
- If you use [Microsoft Purview Managed VNets](#) for metadata ingestion, Managed VNet Runtime and all managed private endpoints will be automatically deployed in the region where your Microsoft Purview is deployed (for example, Australia East).

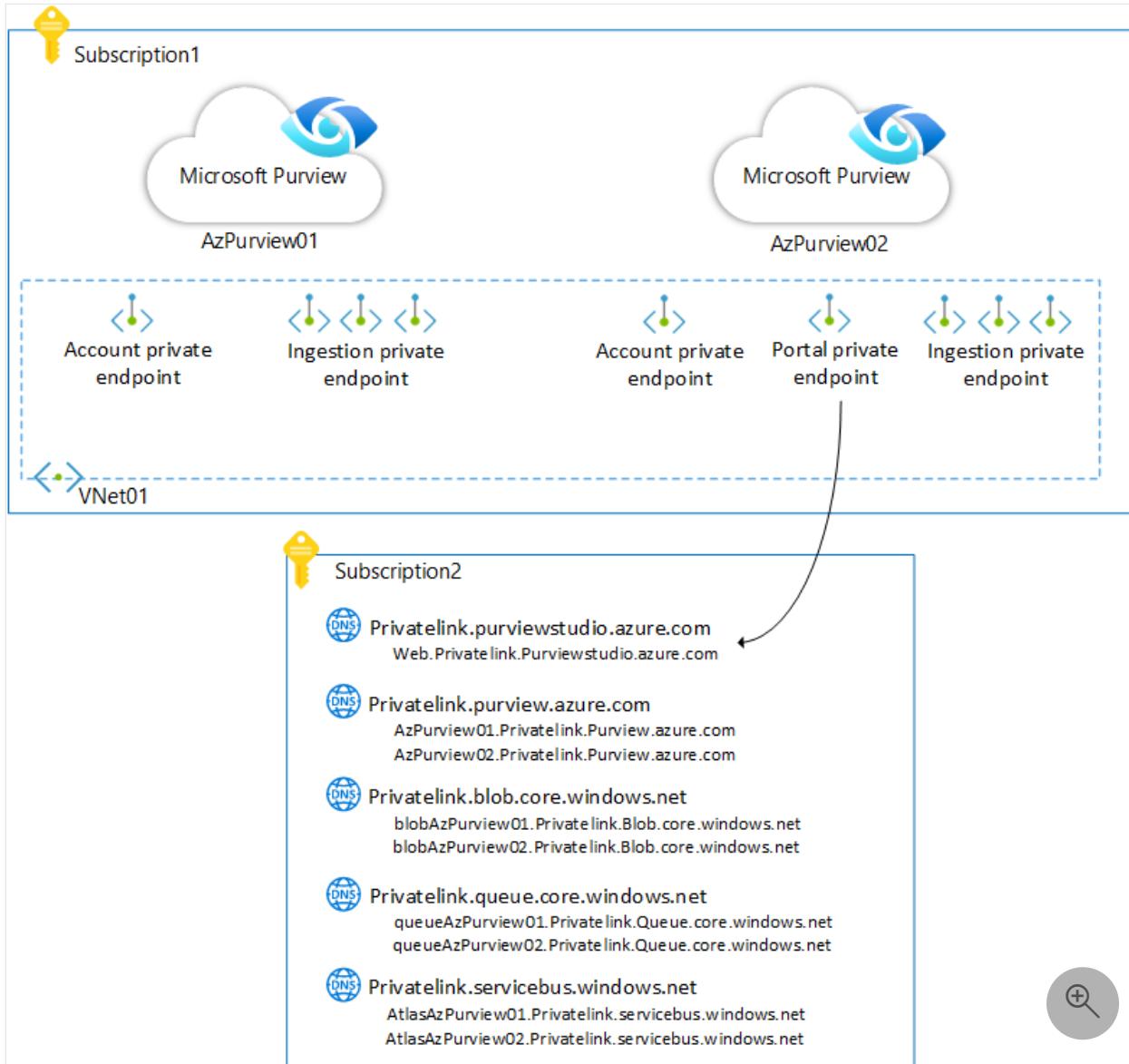
DNS configuration with private endpoints

Name resolution for multiple Microsoft Purview accounts

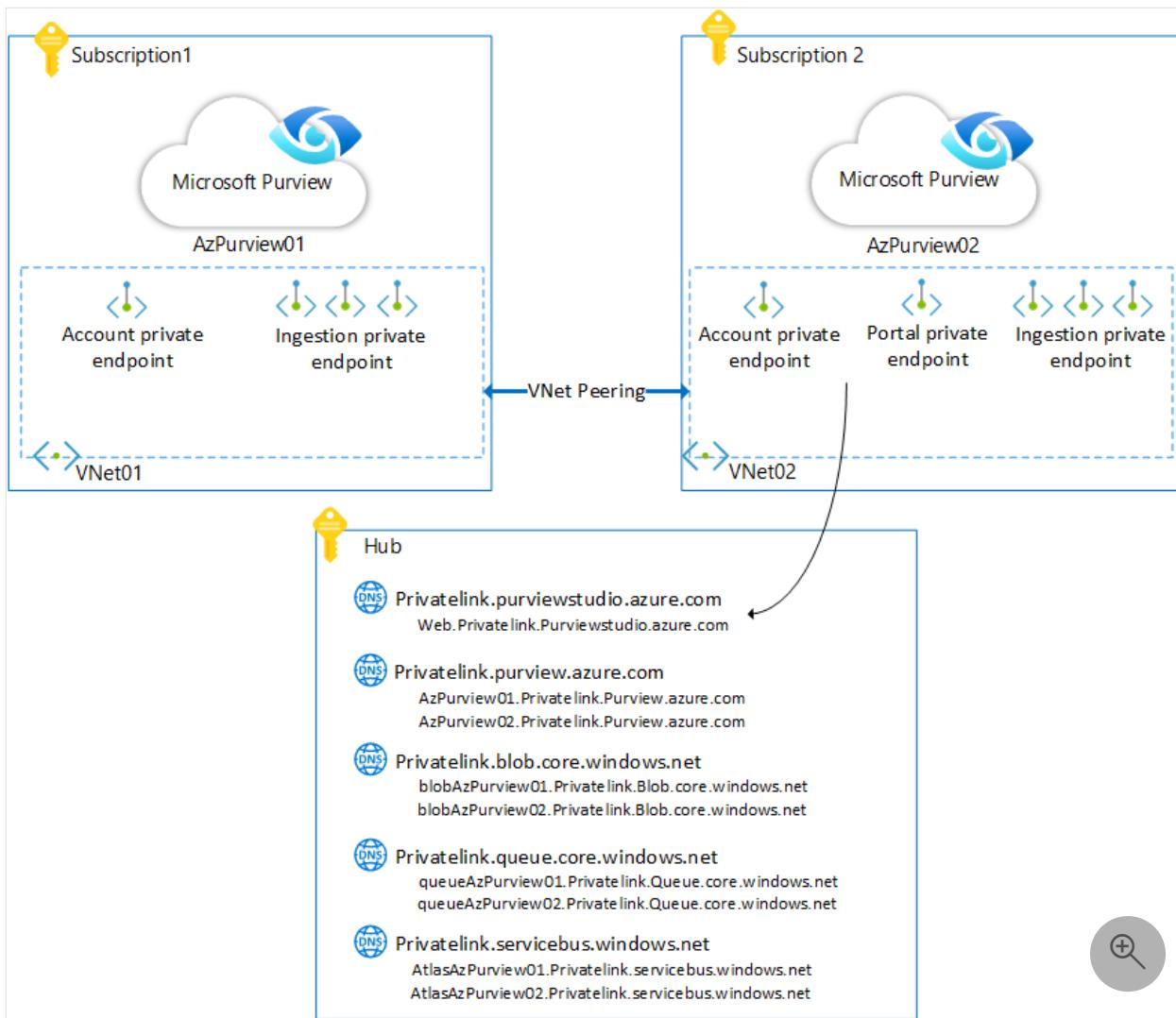
It's recommended to follow these recommendations, if your organization needs to deploy and maintain multiple Microsoft Purview accounts using private endpoints:

1. Deploy at least one *account* private endpoint for each Microsoft Purview account.
2. Deploy at least one set of *ingestion* private endpoints for each Microsoft Purview account.

3. Deploy one *portal* private endpoint for one of the Microsoft Purview accounts in your Azure environments. Create one DNS A record for *portal* private endpoint to resolve `web.purview.azure.com`. The *portal* private endpoint can be used by all purview accounts in the same Azure virtual network or virtual networks connected through VNet peering.



This scenario also applies if multiple Microsoft Purview accounts are deployed across multiple subscriptions and multiple VNets that are connected through VNet peering. *Portal* private endpoint mainly renders static assets related to the Microsoft Purview governance portal, thus, it's independent of Microsoft Purview account, therefore, only one *portal* private endpoint is needed to visit all Microsoft Purview accounts in the Azure environment if VNets are connected.



ⓘ Note

You may need to deploy separate *portal* private endpoints for each Microsoft Purview account in the scenarios where Microsoft Purview accounts are deployed in isolated network segmentations. Microsoft Purview *portal* is static contents for all customers without any customer information. Optionally, you can use public network, (without portal private endpoint) to launch `web.purview.azure.com` if your end users are allowed to launch the Internet.

Option 3: Use both private and public endpoints

You might choose an option in which a subset of your data sources uses private endpoints, and at the same time, you need to scan either of the following:

- Other data sources that are configured with a [service endpoint](#)
- Data sources that have a public endpoint that's accessible through the internet

If you need to scan some data sources by using an ingestion private endpoint and some data sources by using public endpoints or a service endpoint, you can:

1. Use private endpoints for your Microsoft Purview account.
2. Set **Public network access** to **Enabled from all networks** on your Microsoft Purview account.

Integration runtime options

- To scan an Azure data source that's configured with a private endpoint, you need to set up and use a self-hosted integration runtime on a Windows virtual machine that's deployed inside the same or a peered virtual network where Microsoft Purview account and ingestion private endpoints are deployed.

When you're using a private endpoint with Microsoft Purview, you need to allow network connectivity from data sources to a self-hosted integration VM on the Azure virtual network where Microsoft Purview private endpoints are deployed.

- To scan an Azure data source that's configured to allow a public endpoint, you can use the Azure integration runtime.
- To scan on-premises data sources, you can also install a self-hosted integration runtime on either an on-premises Windows machine or a VM inside an Azure virtual network.
- We recommend allowing automatic upgrade for a self-hosted integration runtime. Make sure you open required outbound rules in your Azure virtual network or on your corporate firewall to allow automatic upgrade. For more information, see [Self-hosted integration runtime networking requirements](#).

Authentication options

- To scan an Azure data source that's configured to allow a public endpoint, you can use any authentication option, based on the data source type.
- If you use an ingestion private endpoint to scan an Azure data source that's configured with a private endpoint:
 - You can't use a Microsoft Purview managed identity. Instead, use a service principal, an account key, or SQL authentication, based on the data source type.
 - Make sure that your credentials are stored in an Azure key vault and registered inside Microsoft Purview.

- You must create a credential in Microsoft Purview based on each secret that you create in Azure Key Vault. At minimum, assign *get* and *list* access for secrets for Microsoft Purview on the Key Vault resource in Azure. Otherwise, the credentials won't work in the Microsoft Purview account.

Option 4: Use private endpoints for ingestion only

You might choose this option if you need to:

- Scan all data sources using ingestion private endpoint.
- Managed resources must be configured to disable public network.
- Enable access to Microsoft Purview governance portal through public network.

To enable this option:

1. Configure ingestion private endpoint for your Microsoft Purview account.
2. Set **Public network access** to **Disabled for ingestion only (Preview)** on your [Microsoft Purview account](#).

Integration runtime options

Follow recommendation for option 2.

Authentication options

Follow recommendation for option 2.

Self-hosted integration runtime network and proxy recommendations

For scanning data sources across your on-premises and Azure networks, you may need to deploy and use one or multiple [self-hosted integration runtime virtual machines](#) inside an Azure VNet or an on-premises network, for any of the scenarios mentioned earlier in this document.

- To simplify management, when possible, use Azure IR and [Microsoft Purview Managed VNet IR](#) to scan data sources.
- The Self-hosted integration runtime service can communicate with Microsoft Purview through public or private network over port 443. For more information,

see, [self-hosted integration runtime networking requirements](#).

- One self-hosted integration runtime VM can be used to scan one or multiple data sources in Microsoft Purview, however, self-hosted integration runtime must be only registered for Microsoft Purview and can't be used for Azure Data Factory or Azure Synapse at the same time.
- You can register and use one or multiple self-hosted integration runtimes in one Microsoft Purview account. It's recommended to place at least one self-hosted integration runtime VM in each region or on-premises network where your data sources reside.
- It's recommended to define a baseline for required capacity for each self-hosted integration runtime VM and scale the VM capacity based on demand.
- It's recommended to set up network connection between self-hosted integration runtime VMs and Microsoft Purview and its managed resources through private network, when possible.
- Allow outbound connectivity to download.microsoft.com, if auto-update is enabled.
- The self-hosted integration runtime service doesn't require outbound internet connectivity, if self-hosted integration runtime VMs are deployed in an Azure VNet or in the on-premises network that is connected to Azure through an ExpressRoute or Site to Site VPN connection. In this case, the scan and metadata ingestion process can be done through private network.
- Self-hosted integration runtime can communicate Microsoft Purview and its managed resources directly or through [a proxy server](#). Avoid using proxy settings if self-hosted integration runtime VM is inside an Azure VNet or connected through ExpressRoute or Site to Site VPN connection.
- Review supported scenarios, if you need to use self-hosted integration runtime with [proxy setting](#).

Next steps

- [Use private endpoints for secure access to Microsoft Purview](#)
-

Feedback

Was this page helpful?

 Yes

 No

Overview of pricing for Microsoft Purview (formerly Azure Purview)

Article • 01/12/2024

Microsoft Purview, formerly known as Azure Purview, provides a single pane of glass for managing data governance by enabling automated scanning and classifying data at scale through the Microsoft Purview governance portal.

💡 Tip

For specific price details for governance, see the [Microsoft Purview \(formerly Azure Purview\) pricing page](#). This article will guide you through the features and factors that will affect pricing.

Why do you need to understand the components of pricing?

- While the pricing for Microsoft Purview (formerly Azure Purview) is on a subscription-based **Pay-As-You-Go** model, there are various dimensions that you can consider while budgeting
- This guideline is intended to help you plan the budgeting for Microsoft Purview in the governance portal by providing a view on the control factors that impact the budget

Factors impacting Azure Pricing

There are **direct** and **indirect** costs that need to be considered while planning budgeting and cost management.

Direct costs impacting Microsoft Purview pricing are based on these applications:

- [The Microsoft Purview Data Map](#)
- [Data Estate Insights](#)

Indirect costs

Indirect costs impacting Microsoft Purview (formerly Azure Purview) pricing to be considered are:

- [Managed resources ↗](#)
 - An Event Hubs namespace can be [configured at creation](#) or enabled in the [Azure portal ↗](#) on the Kafka configuration page of the account to enable monitoring with [Atlas Kafka topics events](#). [The Event Hubs will be charged separately ↗](#).
- [Azure private endpoint](#)
 - Azure private end points are used for Microsoft Purview (formerly Azure Purview), where it's required for users on a virtual network (VNet) to securely access the catalog over a private link
 - The [prerequisites](#) for setting up private endpoints could result in extra costs, for example, [costs if you deploy a virtual network ↗](#).
- [Self-hosted integration runtime related costs](#)
 - Self-hosted integration runtime requires infrastructure, which results in extra costs
 - It's required to deploy and register Self-hosted integration runtime (SHIR) inside the same virtual network where Microsoft Purview ingestion private endpoints are deployed
 - [Other memory requirements for scanning](#)
 - Certain data sources such as SAP require more memory on the SHIR machine for scanning
- [Virtual Machine Sizing](#)
 - Plan virtual machine sizing in order to distribute the scanning workload across VMs to optimize the v-cores utilized while running scans
- [Microsoft 365 license](#)
 - Microsoft Purview Information Protection sensitivity labels can be automatically applied to your Azure assets in the Microsoft Purview Data Map.
 - Microsoft Purview Information Protection sensitivity labels are created and managed in the Microsoft Purview compliance portal.
 - To create sensitivity labels for use in Microsoft Purview, you must have an active Microsoft 365 license, which offers the benefit of automatic labeling. For the full list of licenses, see the [Sensitivity labels in Microsoft Purview FAQ](#).
- [Azure Alerts](#)
 - Azure Alerts can notify customers of issues found with infrastructure or applications using the monitoring data in Azure Monitor
 - The pricing for Azure Alerts is available [here ↗](#)

- [Cost Management Budgets & Alerts](#)
 - Automatically generated cost alerts are used in Azure to monitor Azure usage and spending based on when Azure resources are consumed
 - Azure allows you to create and manage Azure budgets. Refer [tutorial](#)
- Multi-cloud egress charges
 - Consider the egress charges (minimal charges added as a part of the multi-cloud subscription) associated with scanning multi-cloud (for example AWS, Google) data sources running native services excepting the S3 and RDS sources

Next steps

- [Microsoft Purview, formerly Azure Purview, pricing page](#)
 - [Pricing guideline Data Estate Insights](#)
 - [Pricing guideline Data Map](#)
-

Feedback

Was this page helpful?

 Yes

 No

Microsoft Purview scanning best practices

Article • 11/02/2023

[Microsoft Purview governance solutions](#) support automated scanning of on-premises, multicloud, and software as a service (SaaS) data sources.

Running a *scan* invokes the process to ingest metadata from the registered data sources. The metadata curated at the end of the scan and curation process includes technical metadata. This metadata can include data asset names such as table names or file names, file size, columns, and data lineage. Schema details are also captured for structured data sources. A relational database management system is an example of this type of source.

The curation process applies automated classification labels on the schema attributes based on the scan rule set configured. Sensitivity labels are applied if your Microsoft Purview account is connected to the Microsoft Purview compliance portal.

Important

If you have any [Azure Policies](#) preventing [updates to Storage accounts](#), this will cause errors for Microsoft Purview's scanning process. Follow the [Microsoft Purview exception tag guide](#) to create an exception for Microsoft Purview accounts.

Why do you need best practices to manage data sources?

Best practices enable you to:

- Optimize cost.
- Build operational excellence.
- Improve security compliance.
- Gain performance efficiency.

Register a source and establish a connection

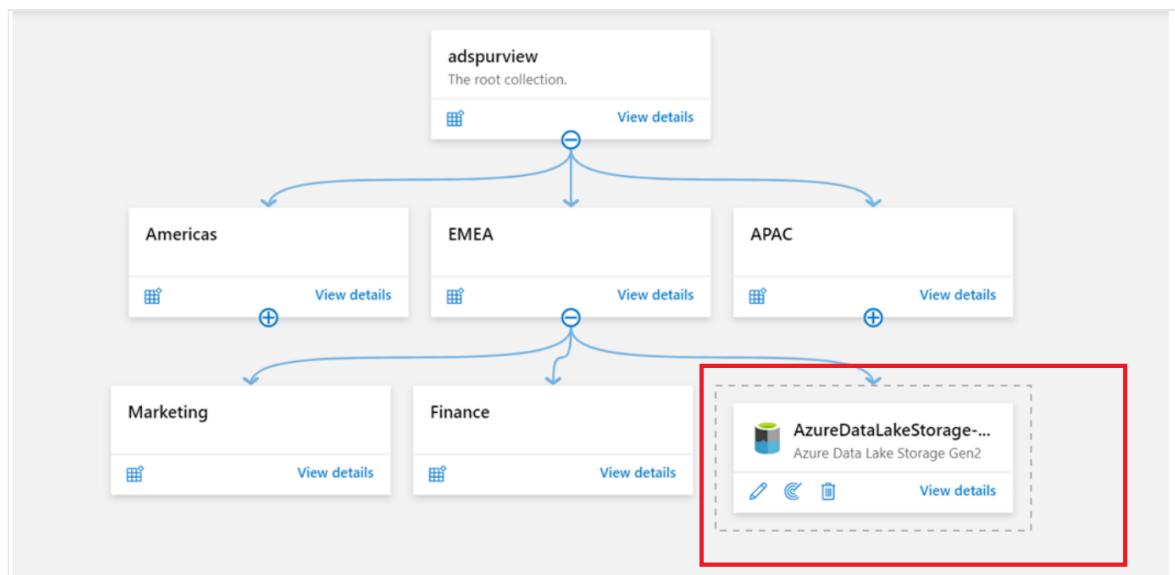
The following design considerations and recommendations help you register a source and establish a connection.

Design considerations

- Use collections to create the hierarchy that aligns with the organization's strategy, like geographical, business function, or source of data. The hierarchy defines the data sources to be registered and scanned.
- By design, you can't register data sources multiple times in the same Microsoft Purview account. This architecture helps to avoid the risk of assigning different access control to the same data source.

Design recommendations

- If the metadata of the same data source is consumed by multiple teams, you can register and manage the data source at a parent collection. Then you can create corresponding scans under each subcollection. In this way, relevant assets appear under each child collection. Sources without parents are grouped in a dotted box in the map view. No arrows link them to parents.



- Use the **Azure Multiple** option if you need to register multiple sources, such as Azure subscriptions or resource groups, in the cloud. For more information, see the following documentation:
 - [Scan multiple sources in Microsoft Purview](#)
 - [Check data source readiness at scale](#)
 - [Configure access to data sources for Microsoft Purview MSI at scale](#)
- After a data source is registered, you might scan the same source multiple times, in case the same source is being used differently by various teams or business units.

For more information on how to define a hierarchy for registering data sources, see [Best practices on collections architecture](#).

Scanning

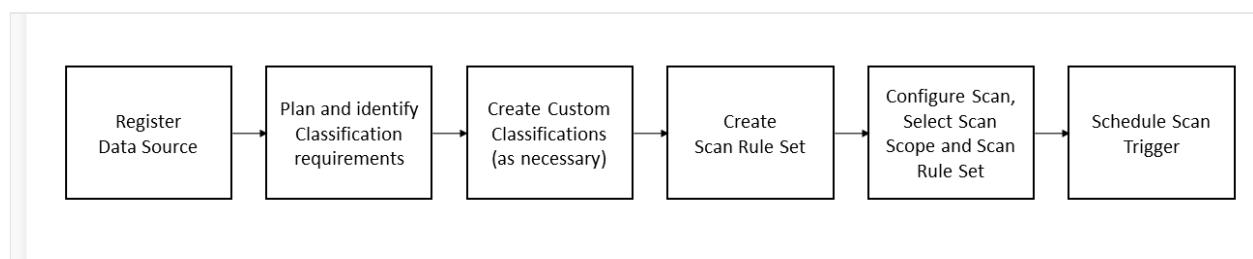
The following design considerations and recommendations are organized based on the key steps involved in the scanning process.

Design considerations

- After the data source is registered, set up a scan to manage automated and secure metadata scanning and curation.
- Scan setup includes configuring the name of the scan, scope of scan, integration runtime, scan trigger frequency, scan rule set, and resource set uniquely for each data source per scan frequency.
- Before you create any credentials, consider your data source types and networking requirements. This information helps you decide which authentication method and integration runtime you need for your scenario.

Design recommendations

After you register your source in the relevant [collection](#), plan and follow the order shown here when you set up the scan. This process order helps you to avoid unexpected costs and rework.



1. Identify your classification requirements from the system in-built classification rules. Or you can create specific custom classification rules, as necessary. Base them on specific industry, business, or regional requirements, which aren't available out of the box:

- See the [classification best practices](#).
- See how to [create a custom classification and classification rule](#).

2. Create scan rule sets before you configure the scan.

Scan rule sets

New Edit Delete Refresh

System Custom

Name

- AdlsGen2
- SqIServer
- AzureStorage
- AzureFileService
- AmazonS3
- AzureMySQL
- AzureSqlDatabase
- AzureCosmosDb
- AzureSqlDatabaseManagedInstance
- AzureDataExplorer

When you create the scan rule set, ensure the following points:

- Verify if the system default scan rule set is sufficient for the data source being scanned. Otherwise, define your custom scan rule set.
- The custom scan rule set can include both system default and custom, so clear those options not relevant for the data assets being scanned.
- Where necessary, create a custom rule set to exclude unwanted classification labels. For example, the system rule set contains generic government code patterns for the planet, not just the United States. Your data might match the pattern of some other type, such as "Belgium Driver's License Number."
- Limit custom classification rules to *most important* and *relevant* labels to avoid clutter. You don't want to have too many labels tagged to the asset.
- If you modify the custom classification or scan rule set, a full scan is triggered. Configure the classification and scan rule set appropriately to avoid rework and costly full scans.

Select classification rules

Choose classification rules that will run on the dataset.

System rules

- > Government
- > Financial
- > Personal
- > Security
- > Miscellaneous

Custom rules

- > Custom classification rules
 - CustomClassDescription
 - ProductID
 - Password_Style_B
 - Password_Style_A
 - NDDDAN
 - testcustom1

[Continue](#)

[Back](#)

193 classification rules selected

[Cancel](#)

! Note

When you scan a storage account, Microsoft Purview uses a set of defined patterns to determine if a group of assets forms a resource set. You can use resource set pattern rules to customize or override how Microsoft Purview detects which assets are grouped as resource sets. The rules also determine how the assets are displayed within the catalog.

For more information, see [Create resource set pattern rules](#). This feature has cost considerations. For information, see the [pricing page](#).

3. Set up a scan for the registered data sources.

- **Scan name:** By default, Microsoft Purview uses the naming convention **SCAN-[A-Z][a-z][a-z]**, which isn't helpful when you're trying to identify a scan that you've run. Be sure to use a meaningful naming convention. For instance, you could name the scan *environment-source-frequency-time* as **DEVODS-Daily-0200**. This name represents a daily scan at 0200 hours.
- **Authentication:** Microsoft Purview offers various authentication methods for scanning data sources, depending on the type of source. It could be Azure cloud or on-premises or third-party sources. Follow the least-privilege principle for the authentication method in this order of preference:
 - Microsoft Purview MSI - Managed Service Identity (for example, for Azure Data Lake Storage Gen2 sources)
 - User-assigned managed identity
 - Service principal
 - SQL authentication (for example, for on-premises or Azure SQL sources)
 - Account key or basic authentication (for example, for SAP S/4HANA sources)

For more information, see the how-to guide to [manage credentials](#).

Note

If you have a firewall enabled for the storage account, you must use the managed identity authentication method when you set up a scan. When you set up a new credential, the credential name can only contain *letters, numbers, underscores, and hyphens*.

• **Integration runtime**

- For more information, see [Network architecture best practices](#).
- If self-hosted integration runtime (SHIR) is deleted, any ongoing scans that rely on it will fail.
- When you use SHIR, make sure that the memory is sufficient for the data source being scanned. For example, when you use SHIR for scanning an SAP source, if you see "out of memory error":
 - Ensure the SHIR machine has enough memory. The recommended amount is 128 GB.

- In the scan setting, set the maximum memory available as some appropriate value, for example, 100.
 - For more information, see the prerequisites in [Scan to and manage SAP ECC Microsoft Purview](#).
- **Scope scan**
 - When you set up the scope for the scan, select only the assets that are relevant at a granular level or parent level. This practice ensures that the scan cost is optimal and performance is efficient. All future assets under a certain parent will be automatically selected if the parent is fully or partially checked.
 - Some examples for some data sources:
 - For Azure SQL Database or Data Lake Storage Gen2, you can scope your scan to specific parts of the data source. Select the appropriate items in the list, such as folders, subfolders, collections, or schemas.
 - For Oracle, Hive Metastore Database, and Teradata sources, a specific list of schemas to be exported can be specified through semicolon-separated values or schema name patterns.
 - For Google Big query, a specific list of datasets to be exported can be specified through semicolon-separated values.
 - When you create a scan for an entire AWS account, you can select specific buckets to scan. When you create a scan for a specific AWS S3 bucket, you can select specific folders to scan.
 - For Erwin, you can scope your scan by providing a semicolon-separated list of Erwin model locator strings.
 - For Cassandra, a specific list of key spaces to be exported can be specified through semicolon-separated values or through key spaces name patterns.
 - For Looker, you can scope your scan by providing a semicolon-separated list of Looker projects.
 - For Power BI tenant, you might only specify whether to include or exclude personal workspace.

Scope your scan

 Refresh

All future assets under a certain parent will be automatically selected if the parent is fully or partially checked.

Search

-  AzureDataLakeStorage-qzO
 -  raw
 - >  BingCoronavirusQuerySet
 - >  Compressed
 - >  Output
 - >  PBIDatasets
 - >  SchemaDrift
 - >  classification01
 - >  testcc
 - >  testcreditcard

Continue

Back

Cancel

- In general, use "ignore patterns," where they're supported, based on wild cards (for example, for data lakes) to exclude temp, config files, RDBMS system tables, or backup or STG tables.
 - When you scan documents or unstructured data, avoid scanning a huge number of such documents. The scan processes the first 20 MB of such documents and might result in longer scan duration.
- **Scan rule set**
 - When you select the scan rule set, make sure to configure the relevant system or custom scan rule set that was created earlier.

- You can create custom filetypes and fill in the details accordingly. Currently, Microsoft Purview supports only one character in Custom Delimiter. If you use custom delimiters, such as ~, in your actual data, you need to create a new scan rule set.

Select a scan rule set

[+ New scan rule set](#) [Refresh](#)

Select one scan rule set to be used by your scan.

 AdlsGen2 <small>SYSTEM DEFAULT</small>	Microsoft default scan rule set that includes all supported file types for schema extraction and classification, and all supported system classification rules View detail
 custom_NDDDAN	Scan any Azure Data Lake Storage Gen2 data. View detail
 Detect_Password	Scan any Azure Data Lake Storage Gen2 data. View detail
 MNCustomScanRS	Scan any Azure Data Lake Storage Gen2 data. View detail
 PasswordDetectionCx	Scan any Azure Data Lake Storage Gen2 data. View detail
 SensitivePIIData	Scan any Azure Data Lake Storage Gen2 data. View detail
 testcompressed	Scan any Azure Data Lake Storage Gen2 data. View detail
 testcreditcard	Scan any Azure Data Lake Storage Gen2 data. View detail

[Continue](#) [Back](#) [Cancel](#)

- **Scan type and schedule**
 - The scan process can be configured to run full or incremental scans.
 - Run the scans during non-business or off-peak hours to avoid any processing overload on the source.

- Initial scan is a full scan, and every subsequent scan is incremental. Subsequent scans can be scheduled as periodic incremental scans. Learn more about the supported [schedule options](#).
- The frequency of scans should align with the change management schedule of the data source or business requirements. For example:
 - If the source structure could potentially change weekly, the scan frequency should be in sync. Changes include new assets or fields within an asset that are added, modified, or deleted.
 - If the classification or sensitivity labels are expected to be up to date on a weekly basis, perhaps for regulatory reasons, the scan frequency should be weekly. For example, if partitions files are added every week in a source data lake, you might schedule monthly scans. You don't need to schedule weekly scans because there's no change in metadata. This suggestion assumes there are no new classification scenarios.
 - The maximum duration that the scan can run is seven days, possibly because of memory issues. This time period excludes the ingestion process. If progress hasn't been updated after seven days, the scan is marked as failed. The ingestion (into catalog) process currently doesn't have any such limitation.
- **Canceling scans**
 - Currently, scans can only be canceled or paused if the status of the scan has transitioned into an "In Progress" state from "Queued" after you trigger the scan.
 - Canceling an individual child scan isn't supported.

Points to note

- If a field or column, table, or a file is removed from the source system after the scan was executed, it will only be reflected (removed) in Microsoft Purview after the next scheduled full or incremental scan.
- An asset can be deleted from a Microsoft Purview catalog by using the **Delete** icon under the name of the asset. This action won't remove the object in the source. If you run a full scan on the same source, it would get reingested in the catalog. If you run incremental scan instead, the deleted asset won't be picked unless the object is modified at the source. An example is if a column is added or removed from the table.
- To understand the behavior of subsequent scans after *manually* editing a data asset or an underlying schema through the Microsoft Purview governance portal, see [Catalog asset details](#).
- For more information, see the tutorial on [how to view, edit, and delete assets](#).

Next steps

[Manage data sources](#)

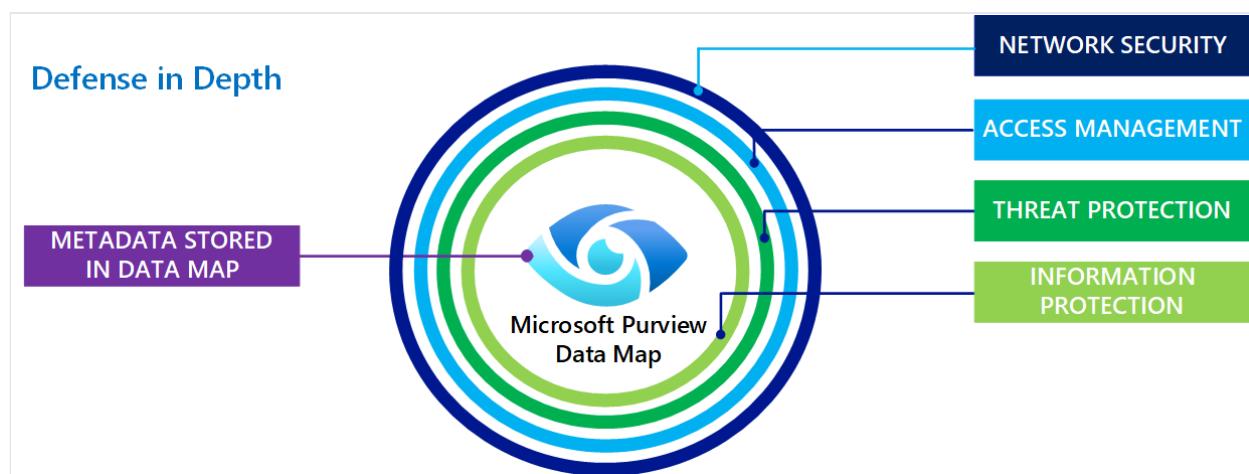
Microsoft Purview security best practices

Article • 07/20/2023

This article provides best practices for common security requirements for Microsoft Purview governance solutions. The security strategy described follows the layered defense-in-depth approach.

ⓘ Note

These best practices cover security for **Microsoft Purview unified governance solutions**. For more information about Microsoft Purview risk and compliance solutions, [go here](#). For more information about Microsoft Purview in general, [go here](#).



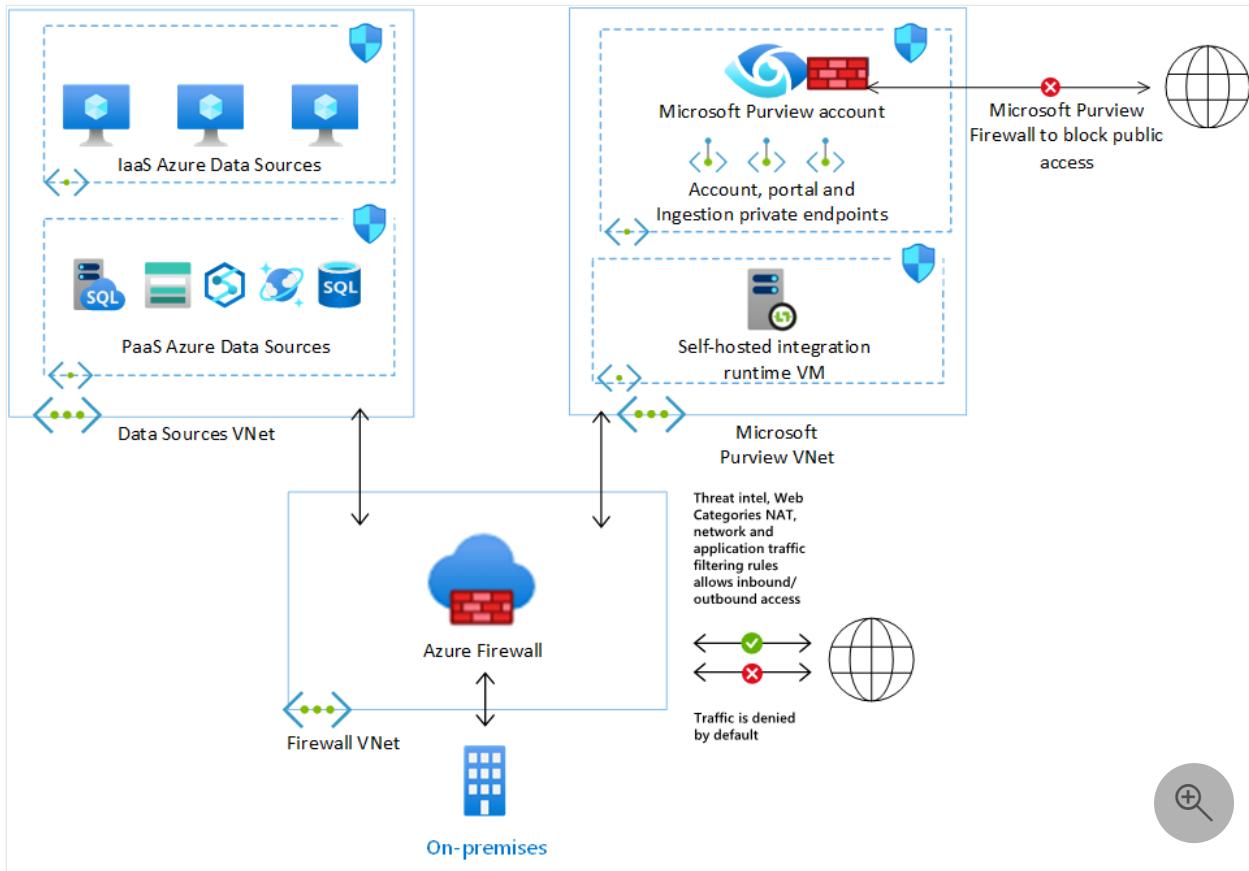
Before applying these recommendations to your environment, you should consult your security team as some may not be applicable to your security requirements.

Network security

Microsoft Purview is a Platform as a Service (PaaS) solution in Azure. You can enable the following network security capabilities for your Microsoft Purview accounts:

- Enable [end-to-end network isolation](#) using Private Link Service.
- Use [Microsoft Purview Firewall](#) to disable Public access.
- Deploy [Network Security Group \(NSG\) rules](#) for subnets where Azure data sources private endpoints, Microsoft Purview private endpoints and self-hosted runtime VMs are deployed.

- Implement Microsoft Purview with private endpoints managed by a Network Virtual Appliance, such as [Azure Firewall](#) for network inspection and network filtering.



For more information, see [Best practices related to connectivity to Azure PaaS Services](#).

Deploy private endpoints for Microsoft Purview accounts

If you need to use Microsoft Purview from inside your private network, it's recommended to use Azure Private Link Service with your Microsoft Purview accounts for partial or [end-to-end isolation](#) to connect to Microsoft Purview governance portal, access Microsoft Purview endpoints and to scan data sources.

The Microsoft Purview *account* private endpoint is used to add another layer of security, so only client calls that are originated from within the virtual network are allowed to access the Microsoft Purview account. This private endpoint is also a prerequisite for the portal private endpoint.

The Microsoft Purview *portal* private endpoint is required to enable connectivity to Microsoft Purview governance portal using a private network.

Microsoft Purview can scan data sources in Azure or an on-premises environment by using ingestion private endpoints.

For more information, see [Microsoft Purview network architecture and best practices](#).

Block public access using Microsoft Purview firewall

You can disable Microsoft Purview Public access to cut off access to the Microsoft Purview account completely from the public internet. In this case, you should consider the following requirements:

- Microsoft Purview must be deployed based on [end-to-end network isolation scenario](#).
- To access Microsoft Purview governance portal and Microsoft Purview endpoints, you need to use a management machine that is connected to private network to access Microsoft Purview through private network.
- Review [known limitations](#).

For more information, see [Firewalls to restrict public access](#).

Use Network Security Groups

You can use an Azure network security group to filter network traffic to and from Azure resources in an Azure virtual network. A network security group contains [security rules](#) that allow or deny inbound network traffic to, or outbound network traffic from, several types of Azure resources. For each rule, you can specify source and destination, port, and protocol.

Network Security Groups can be applied to network interface or Azure virtual networks subnets, where Microsoft Purview private endpoints, self-hosted integration runtime VMs and Azure data sources are deployed.

For more information, see [apply NSG rules for private endpoints](#).

The following NSG rules are required on **data sources** for Microsoft Purview scanning:

Expand table

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
Inbound	Self-hosted integration runtime VMs' private IP	*	Data Sources private IP addresses or Subnets	443	Any	Allow

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
	addresses or subnets					

The following NSG rules are required on from the **management machines** to access Microsoft Purview governance portal:

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

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
Outbound	Management machines' private IP addresses or subnets	*	Microsoft Purview account and portal private endpoint IP addresses or subnets	443	Any	Allow
Outbound	Management machines' private IP addresses or subnets	*	Service tag: AzureCloud	443	Any	Allow

The following NSG rules are required on **self-hosted integration runtime VMs** for Microsoft Purview scanning and metadata ingestion:

ⓘ Important

Consider adding additional rules with relevant Service Tags, based on your data source types.

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

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
Outbound	Self-hosted integration runtime	*	Data Sources private IP addresses or subnets	443	Any	Allow

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
	VMs' private IP addresses or subnets					
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Microsoft Purview account and ingestion private endpoint IP addresses or Subnets	443	Any	Allow
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Service tag: <code>Servicebus</code>	443	Any	Allow
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Service tag: <code>Storage</code>	443	Any	Allow
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Service tag: <code>AzureActiveDirectory</code>	443	Any	Allow
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Service tag: <code>DataFactory</code>	443	Any	Allow
Outbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Service tag: <code>KeyVault</code>	443	Any	Allow

The following NSG rules are required on for Microsoft Purview account, portal and ingestion private endpoints:

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

Direction	Source	Source port range	Destination	Destination port	Protocol	Action
Inbound	Self-hosted integration runtime VMs' private IP addresses or subnets	*	Microsoft Purview account and ingestion private endpoint IP addresses or subnets	443	Any	Allow
Inbound	Management machines' private IP addresses or subnets	*	Microsoft Purview account and ingestion private endpoint IP addresses or subnets	443	Any	Allow

For more information, see [Self-hosted integration runtime networking requirements](#).

Access management

Identity and Access Management provides the basis of a large percentage of security assurance. It enables access based on identity authentication and authorization controls in cloud services. These controls protect data and resources and decide which requests should be permitted.

Related to roles and access management in Microsoft Purview, you can apply the following security best practices:

- Define roles and responsibilities to manage Microsoft Purview in control plane and data plane:
 - Define roles and tasks required to deploy and manage Microsoft Purview inside an Azure subscription.
 - Define roles and task needed to perform data management and governance using Microsoft Purview.
- Assign roles to Microsoft Entra groups instead of assigning roles to individual users.

- Use Azure [Active Directory Entitlement Management](#) to map user access to Microsoft Entra groups using Access Packages.
- Enforce multifactor authentication for Microsoft Purview users, especially, for users with privileged roles such as collection admins, data source admins or data curators.

Manage a Microsoft Purview account in control plane and data plane

Control plane refers to all operations related to Azure deployment and management of Microsoft Purview inside Azure Resource Manager.

Data plane refers to all operations, related to interacting with Microsoft Purview inside Data Map and Data Catalog.

You can assign control plane and data plane roles to users, security groups and service principals from your Microsoft Entra tenant that is associated to Microsoft Purview instance's Azure subscription.

Examples of control plane operations and data plane operations:

[Expand table](#)

Task	Scope	Recommended role	What roles to use?
Deploy a Microsoft Purview account	Control plane	Azure subscription owner or contributor	Azure RBAC roles
Set up a Private Endpoint for Microsoft Purview	Control plane	Contributor	Azure RBAC roles
Delete a Microsoft Purview account	Control plane	Contributor	Azure RBAC roles
Add or manage a self-hosted integration runtime (SHIR)	Control plane	Data source administrator	Microsoft Purview roles
View Microsoft Purview metrics to get current capacity units	Control plane	Reader	Azure RBAC roles
Create a collection	Data plane	Collection Admin	Microsoft Purview roles
Register a data source	Data plane	Collection Admin	Microsoft Purview roles

Task	Scope	Recommended role	What roles to use?
Scan a SQL Server	Data plane	Data source admin and data reader or data curator	Microsoft Purview roles
Search inside Microsoft Purview Data Catalog	Data plane	Data source admin and data reader or data curator	Microsoft Purview roles

Microsoft Purview plane roles are defined and managed inside Microsoft Purview instance in Microsoft Purview collections. For more information, see [Access control in Microsoft Purview](#).

Follow [Azure role-based access recommendations](#) for Azure control plane tasks.

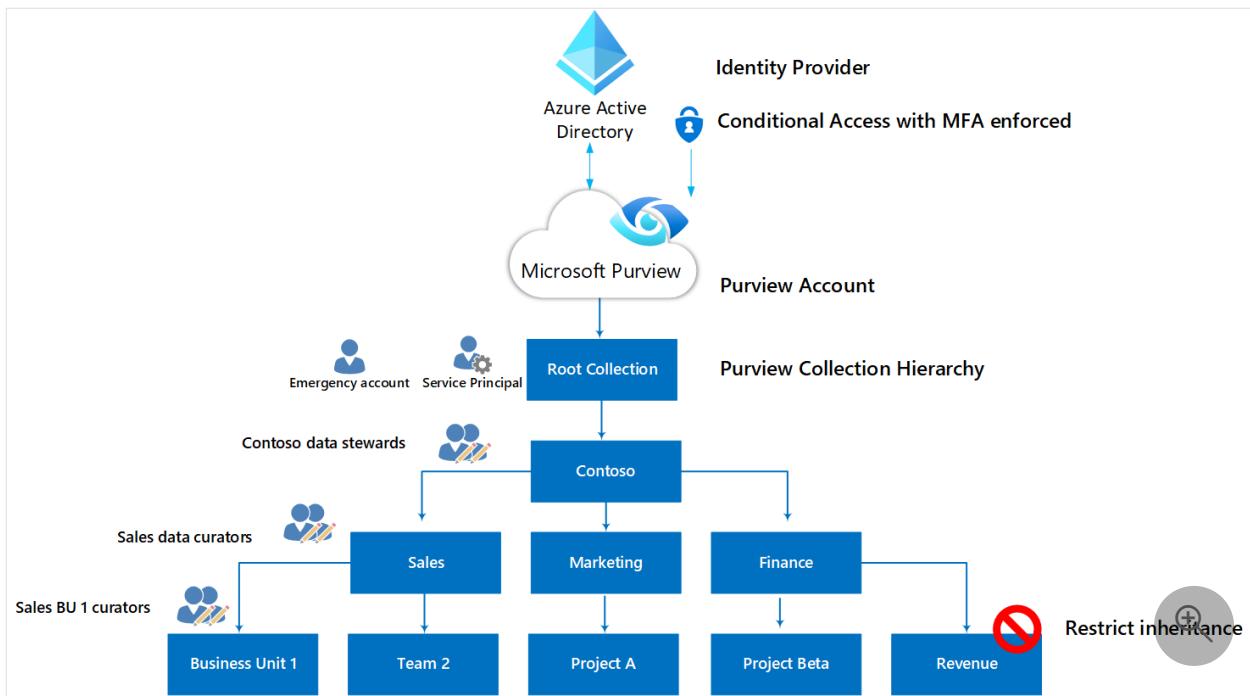
Authentication and authorization

To gain access to Microsoft Purview, users must be authenticated and authorized. Authentication is the process of proving the user is who they claim to be. Authorization refers to controlling access inside Microsoft Purview assigned on collections.

We use Microsoft Entra ID to provide authentication and authorization mechanisms for Microsoft Purview inside Collections. You can assign Microsoft Purview roles to the following security principals from your Microsoft Entra tenant that is associated with Azure subscription where your Microsoft Purview instance is hosted:

- Users and guest users (if they're already added into your Microsoft Entra tenant)
- Security groups
- Managed Identities
- Service Principals

Microsoft Purview fine-grained roles can be assigned to a flexible Collections hierarchy inside the Microsoft Purview instance.



Define Least Privilege model

As a general rule, restrict access based on the [need to know](#) and [least privilege](#) security principles is imperative for organizations that want to enforce security policies for data access.

In Microsoft Purview, data sources, assets and scans can be organized using [Microsoft Purview Collections](#). Collections are hierarchical grouping of metadata in Microsoft Purview, but at the same time they provide a mechanism to manage access across Microsoft Purview. Roles in Microsoft Purview can be assigned to a collection based on your collection's hierarchy.

Use [Microsoft Purview collections](#) to implement your organization's metadata hierarchy for centralized or delegated management and governance hierarchy based on least privileged model.

Follow least privilege access model when assigning roles inside Microsoft Purview collections by segregating duties within your team and grant only the amount of access to users that they need to perform their jobs.

For more information how to assign least privilege access model in Microsoft Purview, based on Microsoft Purview collection hierarchy, see [Access control in Microsoft Purview](#).

Lower exposure of privileged accounts

Securing privileged access is a critical first step to protecting business assets. Minimizing the number of people who have access to secure information or resources, reduces the chance of a malicious user getting access, or an authorized user inadvertently affecting a sensitive resource.

Reduce the number of users with write access inside your Microsoft Purview instance. Keep the number of collection admins and data curator roles minimum at root collection.

Use multifactor authentication and conditional access

[Microsoft Entra multifactor authentication](#) provides another layer of security and authentication. For more security, we recommend enforcing [conditional access policies](#) for all privileged accounts.

By using Microsoft Entra Conditional Access policies, apply Microsoft Entra multifactor authentication at sign-in for all individual users who are assigned to Microsoft Purview roles with modify access inside your Microsoft Purview instances: Collection Admin, Data Source Admin, Data Curator.

Enable multifactor authentication for your admin accounts and ensure that admin account users have registered for MFA.

You can define your Conditional Access policies by selecting Microsoft Purview as a Cloud App.

Prevent accidental deletion of Microsoft Purview accounts

In Azure, you can apply [resource locks](#) to an Azure subscription, a resource group, or a resource to prevent accidental deletion or modification for critical resources.

Enable Azure resource lock for your Microsoft Purview accounts to prevent accidental deletion of Microsoft Purview instances in your Azure subscriptions.

Adding a `CanNotDelete` or `ReadOnly` lock to Microsoft Purview account doesn't prevent deletion or modification operations inside Microsoft Purview data plane, however, it prevents any operations in control plane, such as deleting the Microsoft Purview account, deploying a private endpoint or configuration of diagnostic settings.

For more information, see [Understand scope of locks](#).

Resource locks can be assigned to Microsoft Purview resource groups or resources, however, you can't assign an Azure resource lock to Microsoft Purview Managed resources or managed Resource Group.

Implement a break glass strategy

Plan for a break glass strategy for your Microsoft Entra tenant, Azure subscription and Microsoft Purview accounts to prevent tenant-wide account lockout.

For more information about Microsoft Entra ID and Azure emergency access planning, see [Manage emergency access accounts in Microsoft Entra ID](#).

For more information about Microsoft Purview break glass strategy, see [Microsoft Purview collections best practices and design recommendations](#).

Threat protection and preventing data exfiltration

Microsoft Purview provides rich insights into the sensitivity of your data, which makes it valuable to security teams using Microsoft Defender for Cloud to manage the organization's security posture and protect against threats to their workloads. Data resources remain a popular target for malicious actors, making it crucial for security teams to identify, prioritize, and secure sensitive data resources across their cloud environments. To address this challenge, we're announcing the integration between Microsoft Defender for Cloud and Microsoft Purview in public preview.

Integrate with Microsoft 365 and Microsoft Defender for Cloud

Often, one of the biggest challenges for security organization in a company is to identify and protect assets based on their criticality and sensitivity. Microsoft recently announced [integration between Microsoft Purview and Microsoft Defender for Cloud in Public Preview](#) to help overcome these challenges.

If you've extended your Microsoft 365 sensitivity labels for assets and database columns in Microsoft Purview, you can keep track of highly valuable assets using Microsoft Defender for Cloud from inventory, alerts and recommendations based on assets detected sensitivity labels.

- For recommendations, we've provided **security controls** to help you understand how important each recommendation is to your overall security posture. Microsoft

Defender for Cloud includes a **secure score** value for each control to help you prioritize your security work. Learn more in [Security controls and their recommendations](#).

- For alerts, we've assigned **severity labels** to each alert to help you prioritize the order in which you attend to each alert. Learn more in [How are alerts classified?](#).

For more information, see [Integrate Microsoft Purview with Azure security products](#).

Information Protection

Secure metadata extraction and storage

Microsoft Purview is a data governance solution in cloud. You can register and scan different data sources from various data systems from your on-premises, Azure, or multicloud environments into Microsoft Purview. While data source is registered and scanned in Microsoft Purview, the actual data and data sources stay in their original locations, only metadata is extracted from data sources and stored in Microsoft Purview Data Map, which means you don't need to move data out of the region or their original location to extract the metadata into Microsoft Purview.

If your account was created **before** December 15, 2023, then your account was deployed with a managed resource group and an Azure Storage account was deployed in that group. These resources are consumed by Microsoft Purview and can't be accessed by any other users or principals. An Azure role-based access control (RBAC) deny assignment is added automatically to this resource group when the Microsoft Purview account was deployed, and that prevents access by other users or any CRUD operations that aren't initiated by Microsoft Purview.

Accounts deployed **after** December 15, 2023 (or deployed using API version 2023-05-01-preview onwards) use an ingestion storage account which is deployed in internal Microsoft Azure subscriptions. Since these resources are not on the Microsoft Purview's Azure subscription, they can't be accessed by any other users or principals, except the Microsoft Purview account.

(If you've deployed your accounts using the API, accounts deployed using the API version 2023-05-01-preview onwards will use an ingestion storage account deployed on internal Microsoft Azure subscriptions, instead of a managed storage account deployed on your Azure subscription.)

Where is metadata stored?

Microsoft Purview extracts only the metadata from different data source systems into [Microsoft Purview Data Map](#) during the scanning process.

You can deploy a Microsoft Purview account inside your Azure subscription in any [supported Azure regions](#).

All metadata is stored inside Data Map inside your Microsoft Purview instance. This means the metadata is stored in the same region as your Microsoft Purview instance.

How metadata is extracted from data sources?

Microsoft Purview allows you to use any of the following options to extract metadata from data sources:

- **Azure runtime.** Metadata data is extracted and processed inside the same region as your data sources.
 1. A manual or automatic scan is initiated from the Microsoft Purview Data Map through the Azure integration runtime.
 2. The Azure integration runtime connects to the data source to extract metadata.
 3. Metadata is queued in Microsoft Purview managed or ingestion storage and stored in Azure Blob Storage.
 4. Metadata is sent to the Microsoft Purview Data Map.
- **Self-hosted integration runtime.** Metadata is extracted and processed by self-hosted integration runtime inside self-hosted integration runtime VMs' memory before they're sent to Microsoft Purview Data Map. In this case, customers have to deploy and manage one or more self-hosted integration runtime Windows-based virtual machines inside their Azure subscriptions or on-premises environments. Scanning on-premises and VM-based data sources always requires using a self-hosted integration runtime. The Azure integration runtime isn't supported for these data sources. The following steps show the communication flow at a high level when you're using a self-hosted integration runtime to scan a data source.
 1. A manual or automatic scan is triggered. Microsoft Purview connects to Azure Key Vault to retrieve the credential to access a data source.
 2. The scan is initiated from the Microsoft Purview Data Map through a self-hosted integration runtime.

3. The self-hosted integration runtime service from the VM connects to the data source to extract metadata.
4. Metadata is processed in VM memory for the self-hosted integration runtime. Metadata is queued in Microsoft Purview ingestion storage and then stored in Azure Blob Storage.
5. Metadata is sent to the Microsoft Purview Data Map.

If you need to extract metadata from data sources with sensitive data that can't leave the boundary of your on-premises network, it's highly recommended to deploy the self-hosted integration runtime VM inside your corporate network, where data sources are located, to extract and process metadata in on-premises, and send only metadata to Microsoft Purview.

1. A manual or automatic scan is triggered. Microsoft Purview connects to Azure Key Vault to retrieve the credential to access a data source.
2. The scan is initiated through the on-premises self-hosted integration runtime.
3. The self-hosted integration runtime service from the VM connects to the data source to extract metadata.
4. Metadata is processed in VM memory for the self-hosted integration runtime. Metadata is queued in Microsoft Purview ingestion storage and then stored in Azure Blob Storage. Actual data never leaves the boundary of your network.
5. Metadata is sent to the Microsoft Purview Data Map.

Information protection and encryption

Azure offers many mechanisms for keeping data private in rest and as it moves from one location to another. For Microsoft Purview, data is encrypted at rest using Microsoft-managed keys and when data is in transit, using Transport Layer Security (TLS) v1.2 or greater.

Transport Layer Security (Encryption-in-transit)

Data in transit (also known as data in motion) is encrypted in Microsoft Purview.

To add another layer of security in addition to access controls, Microsoft Purview secures customer data by encrypting data in motion with Transport Layer Security (TLS) and

protect data in transit against 'out of band' attacks (such as traffic capture). It uses encryption to make sure attackers can't easily read or modify the data.

Microsoft Purview supports data encryption in transit with Transport Layer Security (TLS) v1.2 or greater.

For more information, see [Encrypt sensitive information in transit](#).

Transparent data encryption (Encryption-at-rest)

Data at rest includes information that resides in persistent storage on physical media, in any digital format. The media can include files on magnetic or optical media, archived data, and data backups inside Azure regions.

To add another layer of security in addition to access controls, Microsoft Purview encrypts data at rest to protect against 'out of band' attacks (such as accessing underlying storage). It uses encryption with Microsoft-managed keys. This practice helps make sure attackers can't easily read or modify the data.

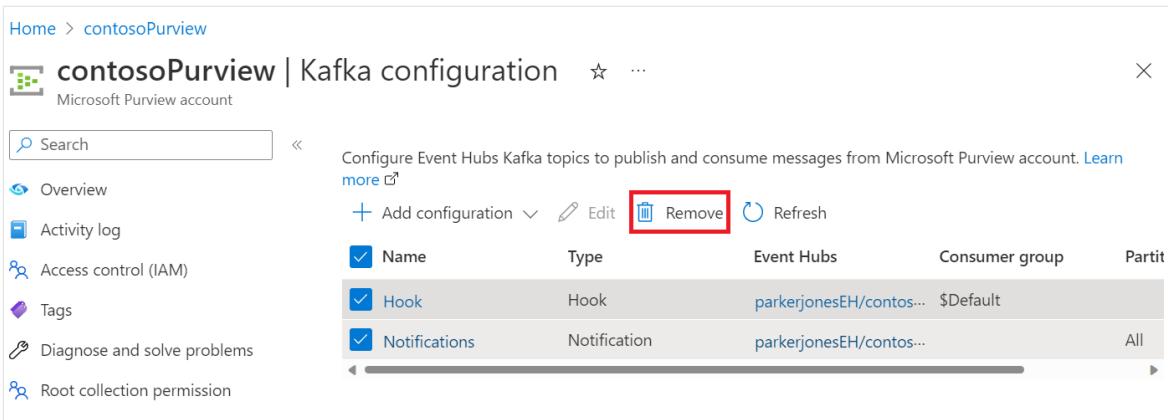
For more information, see [Encrypt sensitive data at rest](#).

Optional Event Hubs namespace configuration

Each Microsoft Purview account can configure Event Hubs that are accessible via their Atlas Kafka endpoint. This can be enabled at creation under *Configuration*, or from the Azure portal under *Kafka configuration*. It's recommended to only enable optional managed event hub if it's used to distribute events into or outside of Microsoft Purview account Data Map. To remove this information distribution point, either don't configure these endpoints, or remove them.

To remove configured Event Hubs namespaces, you can follow these steps:

1. Search for and open your Microsoft Purview account in the [Azure portal](#).
2. Select **Kafka configuration** under settings on your Microsoft Purview account page in the Azure portal.
3. Select the Event Hubs you want to disable. (Hook hubs send messages to Microsoft Purview. Notification hubs receive notifications.)
4. Select **Remove** to save the choice and begin the disablement process. This can take several minutes to complete.



contosoPurview | Kafka configuration

Configure Event Hubs Kafka topics to publish and consume messages from Microsoft Purview account. [Learn more](#)

Name	Type	Event Hubs	Consumer group	Partition
Hook	Hook	parkerjonesEH/contos...	\$Default	All
Notifications	Notification	parkerjonesEH/contos...		All

ⓘ Note

If you have an ingestion private endpoint when you disable this Event Hubs namespace, after disabling the ingestion private endpoint may show as disconnected.

For more information about configuring these Event Hubs namespaces, see: [Configure Event Hubs for Atlas Kafka topics](#)

Credential management

To extract metadata from a data source system into Microsoft Purview Data Map, it's required to register and scan the data source systems in Microsoft Purview Data Map. To automate this process, we have made available [connectors](#) for different data source systems in Microsoft Purview to simplify the registration and scanning process.

To connect to a data source Microsoft Purview requires a credential with read-only access to the data source system.

It's recommended prioritizing the use of the following credential options for scanning, when possible:

1. Microsoft Purview Managed Identity
2. User Assigned Managed Identity
3. Service Principals
4. Other options such as Account key, SQL Authentication, etc.

If you use any options rather than managed identities, all credentials must be stored and protected inside an [Azure key vault](#). Microsoft Purview requires get/list access to secret on the Azure Key Vault resource.

As a general rule, you can use the following options to set up integration runtime, and credentials to scan data source systems:

[] [Expand table](#)

Scenario	Runtime option	Supported Credentials
Data source is an Azure Platform as a Service, such as Azure Data Lake Storage Gen 2 or Azure SQL inside public network	Option 1: Azure Runtime	Microsoft Purview Managed Identity, Service Principal or Access Key / SQL Authentication (depending on Azure data source type)
Data source is an Azure Platform as a Service, such as Azure Data Lake Storage Gen 2 or Azure SQL inside public network	Option 2: Self-hosted integration runtime	Service Principal or Access Key / SQL Authentication (depending on Azure data source type)
Data source is an Azure Platform as a Service, such as Azure Data Lake Storage Gen 2 or Azure SQL inside private network using Azure Private Link Service	Self-hosted integration runtime	Service Principal or Access Key / SQL Authentication (depending on Azure data source type)
Data source is inside an Azure IaaS VM such as SQL Server	Self-hosted integration runtime deployed in Azure	SQL Authentication or Basic Authentication (depending on Azure data source type)
Data source is inside an on-premises system such as SQL Server or Oracle	Self-hosted integration runtime deployed in Azure or in the on-premises network	SQL Authentication or Basic Authentication (depending on Azure data source type)
Multicloud	Azure runtime or self-hosted integration runtime based on data source types	Supported credential options vary based on data sources types
Power BI tenant	Azure Runtime	Microsoft Purview Managed Identity

Use [this guide](#) to read more about each source and their supported authentication options.

Other recommendations

Apply security best practices for Self-hosted runtime VMs

Consider securing the deployment and management of self-hosted integration runtime VMs in Azure or your on-premises environment, if self-hosted integration runtime is used to scan data sources in Microsoft Purview.

For self-hosted integration runtime VMs deployed as virtual machines in Azure, follow [security best practices recommendations for Windows virtual machines](#).

- Lock down inbound traffic to your VMs using Network Security Groups and [Azure Defender access Just-in-Time](#).
- Install antivirus or antimalware.
- Deploy Azure Defender to get insights around any potential anomaly on the VMs.
- Limit the amount of software in the self-hosted integration runtime VMs. Although it isn't a mandatory requirement to have a dedicated VM for a self-hosted runtime for Microsoft Purview, we highly suggest using dedicated VMs especially for production environments.
- Monitor the VMs using [Azure Monitor for VMs](#). By using Log analytics agent, you can capture content such as performance metrics to adjust required capacity for your VMs.
- By integrating virtual machines with Microsoft Defender for Cloud, you can prevent, detect, and respond to threats.
- Keep your machines current. You can enable Automatic Windows Update or use [Update Management in Azure Automation](#) to manage operating system level updates for the OS.
- Use multiple machines for greater resilience and availability. You can deploy and register multiple self-hosted integration runtimes to distribute the scans across multiple self-hosted integration runtime machines or deploy the self-hosted integration runtime on a Virtual Machine Scale Set for higher redundancy and scalability.
- Optionally, you can plan to enable Azure backup from your self-hosted integration runtime VMs to increase the recovery time of a self-hosted integration runtime VM if there's a VM level disaster.

Next steps

- [Microsoft Purview network architecture and best practices](#)
- [Credentials for source authentication in Microsoft Purview](#)

Feedback

Was this page helpful?

 Yes

 No

Continuous integration and delivery in Azure Data Factory

Article • 10/26/2023

APPLIES TO:  Azure Data Factory  Azure Synapse Analytics

Tip

Try out **Data Factory in Microsoft Fabric**, an all-in-one analytics solution for enterprises. **Microsoft Fabric** covers everything from data movement to data science, real-time analytics, business intelligence, and reporting. Learn how to [start a new trial](#) for free!

Continuous integration is the practice of testing each change made to your codebase automatically and as early as possible. Continuous delivery follows the testing that happens during continuous integration and pushes changes to a staging or production system.

In Azure Data Factory, continuous integration and delivery (CI/CD) means moving Data Factory pipelines from one environment (development, test, production) to another. Azure Data Factory utilizes [Azure Resource Manager templates](#) to store the configuration of your various ADF entities (pipelines, datasets, data flows, and so on). There are two suggested methods to promote a data factory to another environment:

- Automated deployment using Data Factory's integration with [Azure Pipelines](#)
- Manually upload a Resource Manager template using Data Factory UX integration with Azure Resource Manager.

Note

We recommend that you use the Azure Az PowerShell module to interact with Azure. See [Install Azure PowerShell](#) to get started. To learn how to migrate to the Az PowerShell module, see [Migrate Azure PowerShell from AzureRM to Az.](#)

CI/CD lifecycle

Note

For more information, see [Continuous deployment improvements](#).

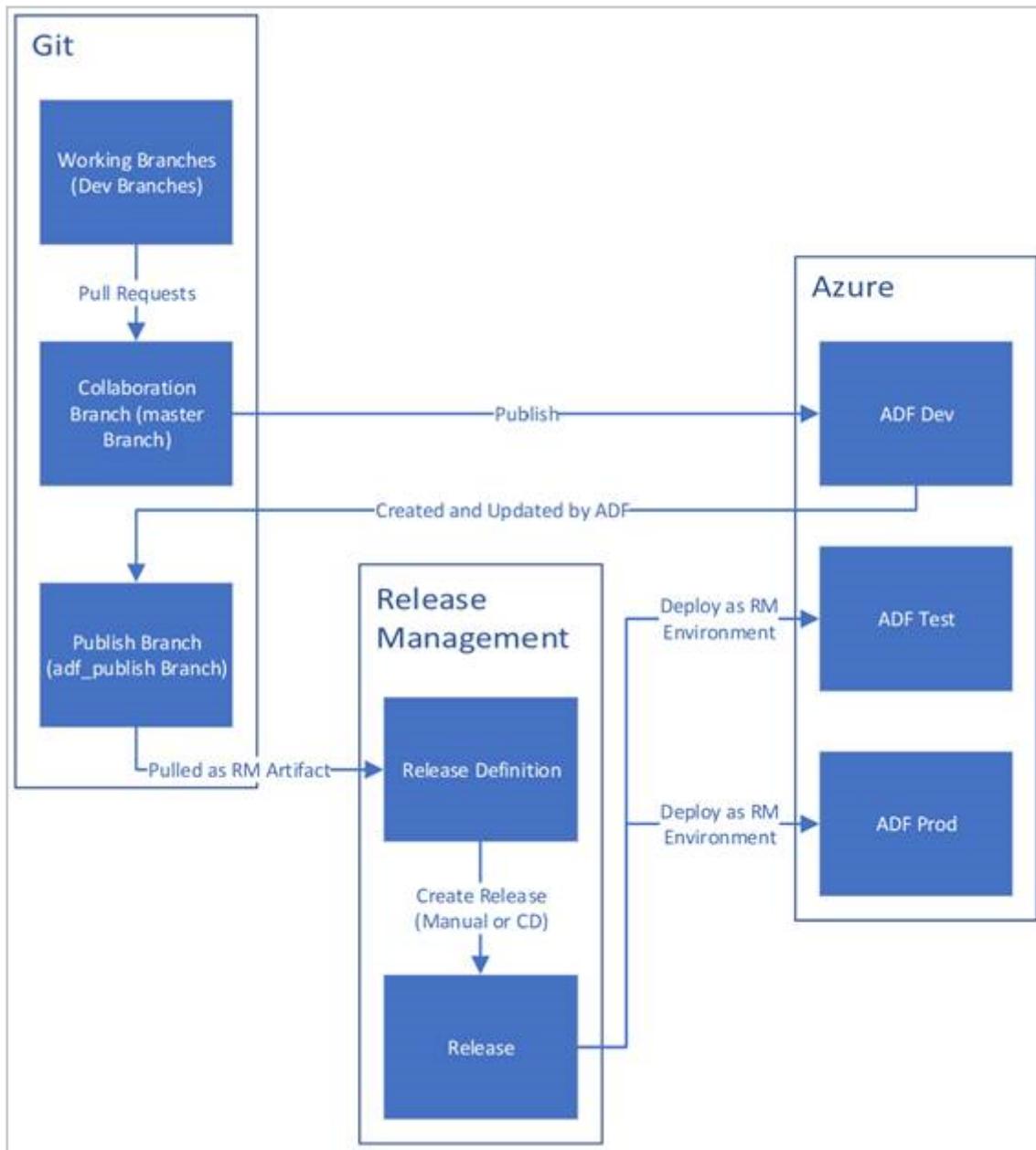
Below is a sample overview of the CI/CD lifecycle in an Azure data factory that's configured with Azure Repos Git. For more information on how to configure a Git repository, see [Source control in Azure Data Factory](#).

1. A development data factory is created and configured with Azure Repos Git. All developers should have permission to author Data Factory resources like pipelines and datasets.
2. A developer [creates a feature branch](#) to make a change. They debug their pipeline runs with their most recent changes. For more information on how to debug a pipeline run, see [Iterative development and debugging with Azure Data Factory](#).
3. After a developer is satisfied with their changes, they create a pull request from their feature branch to the main or collaboration branch to get their changes reviewed by peers.
4. After a pull request is approved and changes are merged in the main branch, the changes get published to the development factory.
5. When the team is ready to deploy the changes to a test or UAT (User Acceptance Testing) factory, the team goes to their Azure Pipelines release and deploys the desired version of the development factory to UAT. This deployment takes place as part of an Azure Pipelines task and uses Resource Manager template parameters to apply the appropriate configuration.
6. After the changes have been verified in the test factory, deploy to the production factory by using the next task of the pipelines release.

 **Note**

Only the development factory is associated with a git repository. The test and production factories shouldn't have a git repository associated with them and should only be updated via an Azure DevOps pipeline or via a Resource Management template.

The below image highlights the different steps of this lifecycle.



Best practices for CI/CD

If you're using Git integration with your data factory and have a CI/CD pipeline that moves your changes from development into test and then to production, we recommend these best practices:

- **Git integration.** Configure only your development data factory with Git integration. Changes to test and production are deployed via CI/CD and don't need Git integration.
- **Pre- and post-deployment script.** Before the Resource Manager deployment step in CI/CD, you need to complete certain tasks, like stopping and restarting triggers and performing cleanup. We recommend that you use PowerShell scripts before and after the deployment task. For more information, see [Update active triggers](#).

The data factory team has [provided a script](#) to use located at the bottom of this page.

 **Note**

Use the [PrePostDeploymentScript.Ver2.ps1](#) if you would like to turn off/ on only the triggers that have been modified instead of turning all triggers off/ on during CI/CD.

 **Warning**

Make sure to use **PowerShell Core** in ADO task to run the script.

 **Warning**

If you do not use latest versions of PowerShell and Data Factory module, you may run into deserialization errors while running the commands.

- **Integration runtimes and sharing.** Integration runtimes don't change often and are similar across all stages in your CI/CD. So Data Factory expects you to have the same name, type and sub-type of integration runtime across all stages of CI/CD. If you want to share integration runtimes across all stages, consider using a ternary factory just to contain the shared integration runtimes. You can use this shared factory in all of your environments as a linked integration runtime type.

 **Note**

The integration runtime sharing is only available for self-hosted integration runtimes. Azure-SSIS integration runtimes don't support sharing.

- **Managed private endpoint deployment.** If a private endpoint already exists in a factory and you try to deploy an ARM template that contains a private endpoint with the same name but with modified properties, the deployment will fail. In other words, you can successfully deploy a private endpoint as long as it has the same properties as the one that already exists in the factory. If any property is different between environments, you can override it by parameterizing that property and providing the respective value during deployment.

- **Key Vault.** When you use linked services whose connection information is stored in Azure Key Vault, it is recommended to keep separate key vaults for different environments. You can also configure separate permission levels for each key vault. For example, you might not want your team members to have permissions to production secrets. If you follow this approach, we recommend that you to keep the same secret names across all stages. If you keep the same secret names, you don't need to parameterize each connection string across CI/CD environments because the only thing that changes is the key vault name, which is a separate parameter.
- **Resource naming.** Due to ARM template constraints, issues in deployment may arise if your resources contain spaces in the name. The Azure Data Factory team recommends using '_' or '-' characters instead of spaces for resources. For example, 'Pipeline_1' would be a preferable name over 'Pipeline 1'.
- **Altering repository.** ADF manages GIT repository content automatically. Altering or adding manually unrelated files or folder into anywhere in ADF Git repository data folder could cause resource loading errors. For example, presence of `.bak` files can cause ADF CI/CD error, so they should be removed for ADF to load.
- **Exposure control and feature flags.** When working in a team, there are instances where you may merge changes, but don't want them to be run in elevated environments such as PROD and QA. To handle this scenario, the ADF team recommends [the DevOps concept of using feature flags](#). In ADF, you can combine [global parameters](#) and the [if condition activity](#) to hide sets of logic based upon these environment flags.

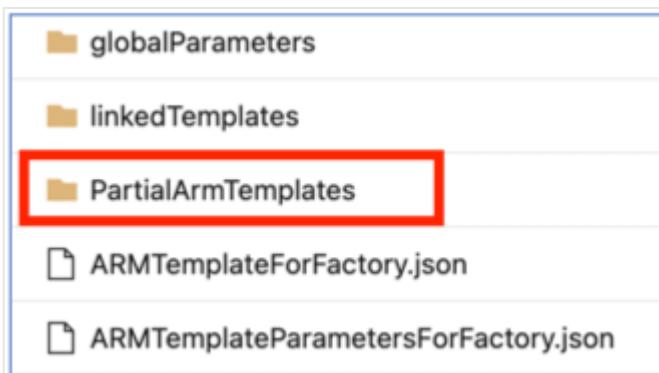
To learn how to set up a feature flag, see the below video tutorial:

<https://www.microsoft.com/en-us/videoplayer/embed/RE4IxdW?postJs||Msg=true> ↗

Unsupported features

- By design, Data Factory doesn't allow cherry-picking of commits or selective publishing of resources. Publishes will include all changes made in the data factory.
 - Data factory entities depend on each other. For example, triggers depend on pipelines, and pipelines depend on datasets and other pipelines. Selective publishing of a subset of resources could lead to unexpected behaviors and errors.
 - On rare occasions when you need selective publishing, consider using a hotfix. For more information, see [Hotfix production environment](#).

- The Azure Data Factory team doesn't recommend assigning Azure RBAC controls to individual entities (pipelines, datasets, etc.) in a data factory. For example, if a developer has access to a pipeline or a dataset, they should be able to access all pipelines or datasets in the data factory. If you feel that you need to implement many Azure roles within a data factory, look at deploying a second data factory.
- You can't publish from private branches.
- You can't currently host projects on Bitbucket.
- You can't currently export and import alerts and matrices as parameters.
- In the code repository under the *adf_publish* branch, a folder named 'PartialArmTemplates' is currently added beside the 'linkedTemplates' folder, 'ARMTemplateForFactory.json' and 'ARMTemplateParametersForFactory.json' files as part of publishing with source control.



We will no longer be publishing 'PartialArmTemplates' to the *adf_publish* branch starting 1-November 2021.

No action is required unless you are using 'PartialArmTemplates'. Otherwise, switch to any supported mechanism for deployments using: 'ARMTemplateForFactory.json' or 'linkedTemplates' files.

Next steps

- [Continuous deployment improvements](#)
- [Automate continuous integration using Azure Pipelines releases](#)
- [Manually promote a Resource Manager template to each environment](#)
- [Use custom parameters with a Resource Manager template](#)
- [Linked Resource Manager templates](#)
- [Using a hotfix production environment](#)
- [Sample pre- and post-deployment script](#)

Automated publishing for continuous integration and delivery

Article • 12/04/2023

APPLIES TO:  Azure Data Factory  Azure Synapse Analytics

Tip

Try out **Data Factory in Microsoft Fabric**, an all-in-one analytics solution for enterprises. **Microsoft Fabric** covers everything from data movement to data science, real-time analytics, business intelligence, and reporting. Learn how to [start a new trial](#) for free!

Overview

Continuous integration is the practice of testing each change made to your codebase automatically. As early as possible, continuous delivery follows the testing that happens during continuous integration and pushes changes to a staging or production system.

In Azure Data Factory, continuous integration and continuous delivery (CI/CD) means moving Data Factory pipelines from one environment, such as development, test, and production, to another. Data Factory uses [Azure Resource Manager templates \(ARM templates\)](#) to store the configuration of your various Data Factory entities, such as pipelines, datasets, and data flows.

There are two suggested methods to promote a data factory to another environment:

- Automated deployment using the integration of Data Factory with [Azure Pipelines](#).
- Manually uploading an ARM template by using Data Factory user experience integration with Azure Resource Manager.

For more information, see [Continuous integration and delivery in Azure Data Factory](#).

This article focuses on the continuous deployment improvements and the automated publish feature for CI/CD.

Continuous deployment improvements

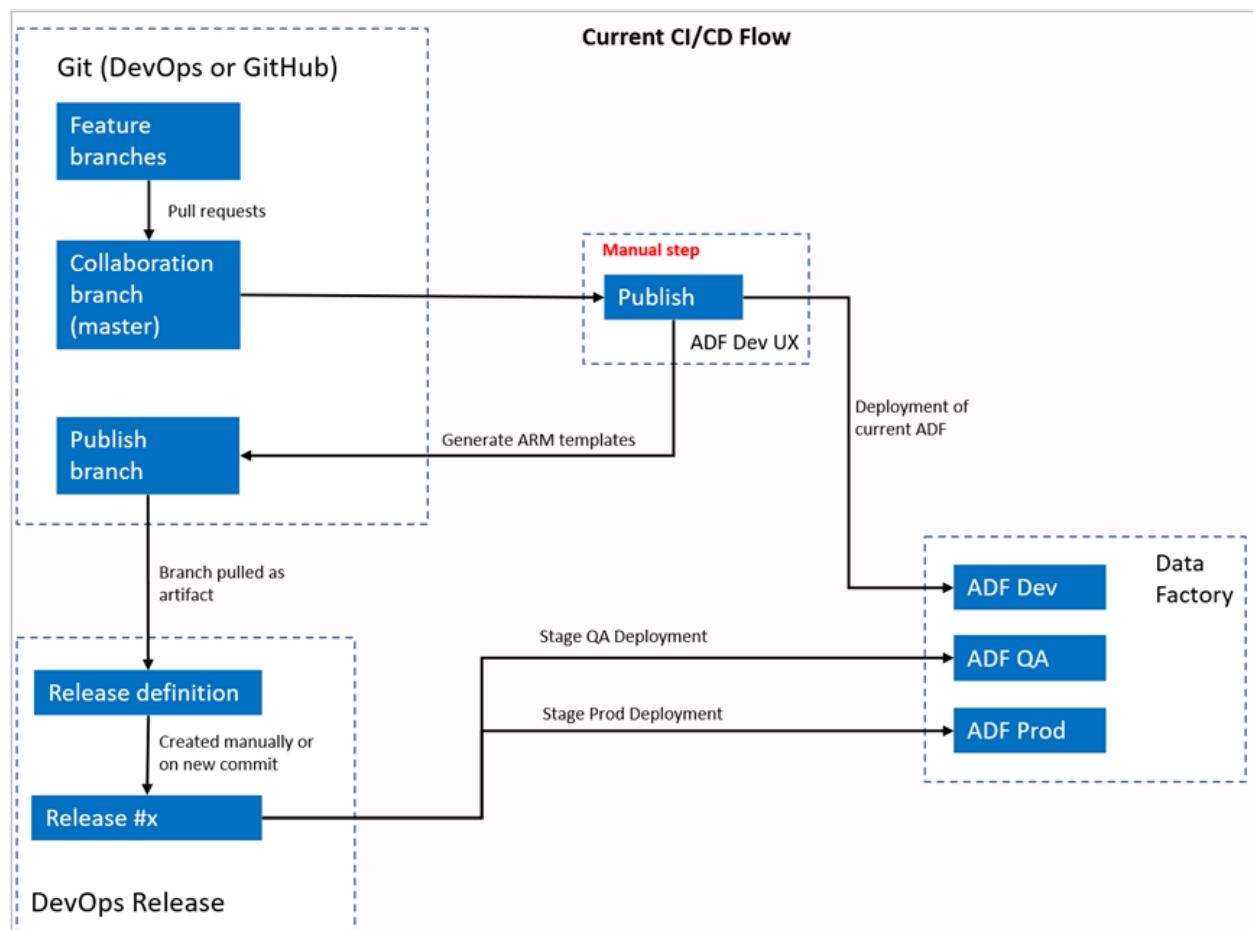
The automated publish feature takes the **Validate all** and **Export ARM template** features from the Data Factory user experience and makes the logic consumable via a publicly available npm package [@microsoft/azure-data-factory-utilities](#) . For this reason, you can programmatically trigger these actions instead of having to go to the Data Factory UI and select a button manually. This capability will give your CI/CD pipelines a truer continuous integration experience.

Note

Be sure to use the latest versions of node and npm to avoid errors that can occur due to package incompatibility with older versions.

Current CI/CD flow

1. Each user makes changes in their private branches.
2. Push to master isn't allowed. Users must create a pull request to make changes.
3. Users must load the Data Factory UI and select **Publish** to deploy changes to Data Factory and generate the ARM templates in the publish branch.
4. The DevOps Release pipeline is configured to create a new release and deploy the ARM template each time a new change is pushed to the publish branch.

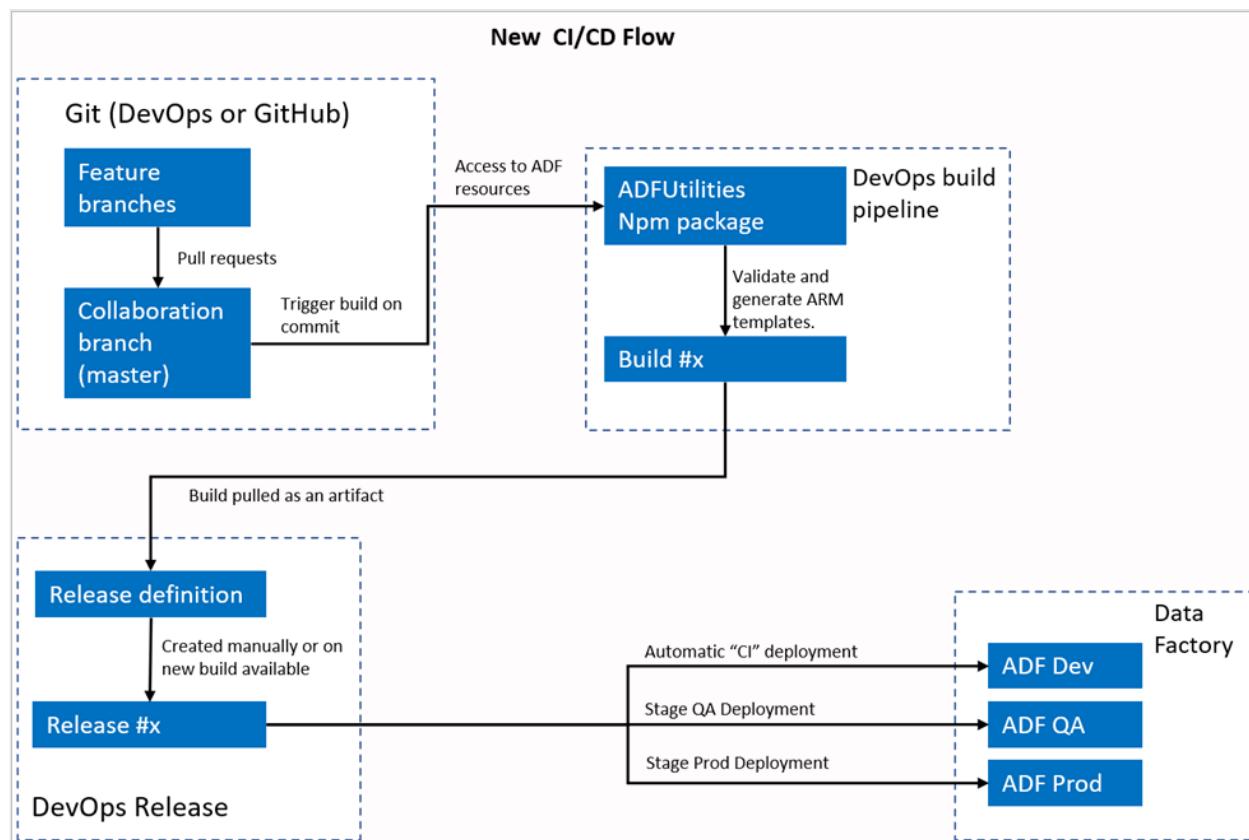


Manual step

In the current CI/CD flow, the user experience is the intermediary to create the ARM template. As a result, a user must go to the Data Factory UI and manually select **Publish** to start the ARM template generation and drop it in the publish branch.

The new CI/CD flow

1. Each user makes changes in their private branches.
2. Push to master isn't allowed. Users must create a pull request to make changes.
3. The Azure DevOps pipeline build is triggered every time a new commit is made to master. It validates the resources and generates an ARM template as an artifact if validation succeeds.
4. The DevOps Release pipeline is configured to create a new release and deploy the ARM template each time a new build is available.



What changed?

- We now have a build process that uses a DevOps build pipeline.
- The build pipeline uses the ADFUtilities NPM package, which will validate all the resources and generate the ARM templates. These templates can be single and linked.

- The build pipeline is responsible for validating Data Factory resources and generating the ARM template instead of the Data Factory UI (Publish button).
- The DevOps release definition will now consume this new build pipeline instead of the Git artifact.

ⓘ Note

You can continue to use the existing mechanism, which is the `adf_publish` branch, or you can use the new flow. Both are supported.

Package overview

Two commands are currently available in the package:

- Export ARM template
- Validate

Export ARM template

Run `npm run build export <rootFolder> <factoryId> [outputFolder]` to export the ARM template by using the resources of a given folder. This command also runs a validation check prior to generating the ARM template. Here's an example using a resource group named `testResourceGroup`:

dos

```
npm run build export C:\DataFactories\DevDataFactory
/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourceGroups/testResourceGroup/providers/Microsoft.DataFactor
y/factories/DevDataFactory ArmTemplateOutput
```

- `RootFolder` is a mandatory field that represents where the Data Factory resources are located.
- `FactoryId` is a mandatory field that represents the Data Factory resource ID in the format `/subscriptions/<subId>/resourceGroups/<rgName>/providers/Microsoft.DataFactory
/factories/<dfName>`.
- `OutputFolder` is an optional parameter that specifies the relative path to save the generated ARM template.

The ability to stop/start only the updated triggers is now generally available and is merged into the command shown above.

ⓘ Note

The ARM template generated isn't published to the live version of the factory. Deployment should be done by using a CI/CD pipeline.

Validate

Run `npm run build validate <rootFolder> <factoryId>` to validate all the resources of a given folder. Here's an example:

dos

```
npm run build validate C:\DataFactories\DevDataFactory
/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourceGroups/testResourceGroup/providers/Microsoft.DataFactor
y/factories/DevDataFactory
```

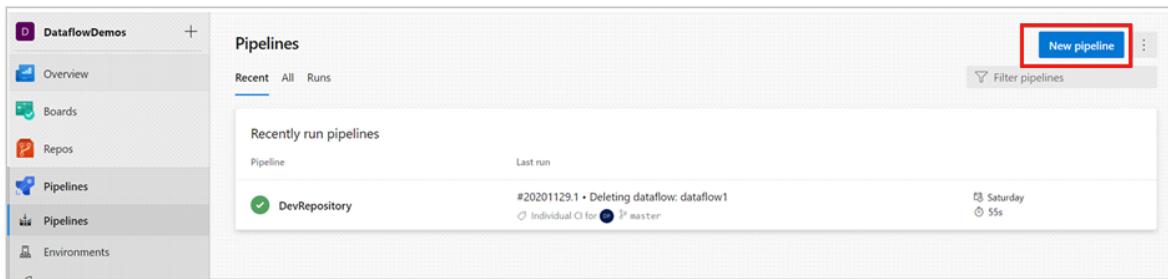
- `RootFolder` is a mandatory field that represents where the Data Factory resources are located.
- `FactoryId` is a mandatory field that represents the Data Factory resource ID in the format
`/subscriptions/<subId>/resourceGroups/<rgName>/providers/Microsoft.DataFactory
/factories/<dfName>.`

Create an Azure pipeline

While npm packages can be consumed in various ways, one of the primary benefits is being consumed via [Azure Pipeline](#). On each merge into your collaboration branch, a pipeline can be triggered that first validates all of the code and then exports the ARM template into a [build artifact](#) that can be consumed by a release pipeline. How it differs from the current CI/CD process is that you will *point your release pipeline at this artifact instead of the existing `adf_publish` branch*.

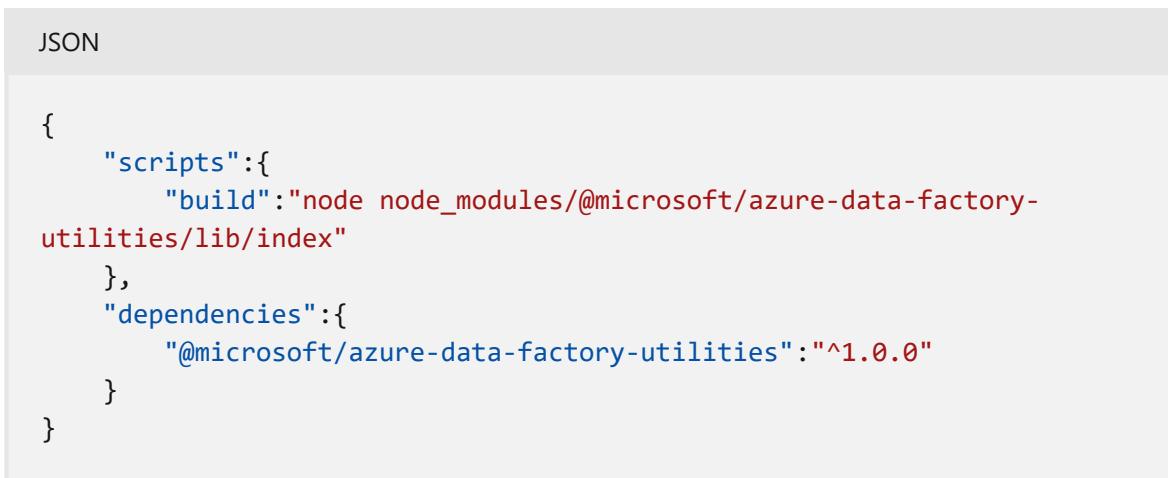
Follow these steps to get started:

1. Open an Azure DevOps project, and go to [Pipelines](#). Select [New Pipeline](#).



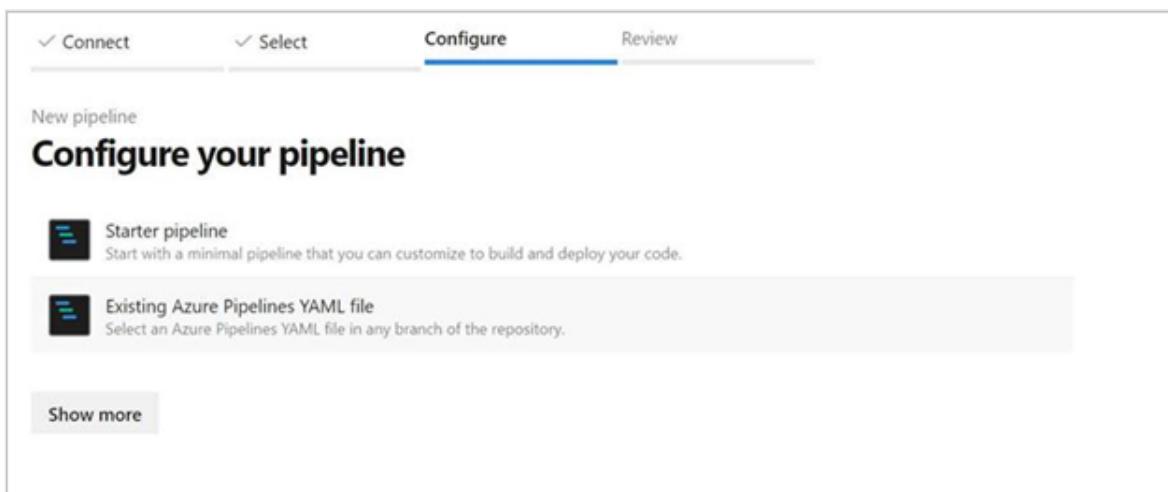
The screenshot shows the 'Pipelines' blade in the Azure Data Factory interface. On the left, there's a sidebar with 'DataflowDemos' selected. The main area shows a table of 'Recently run pipelines'. One row is highlighted for 'DevRepository'. At the top right, there's a 'New pipeline' button with a red box around it. Below the table, there's a 'Filter pipelines' search bar and a three-dot menu icon.

2. Select the repository where you want to save your pipeline YAML script. We recommend saving it in a build folder in the same repository of your Data Factory resources. Ensure there's a *package.json* file in the repository that contains the package name, as shown in the following example:

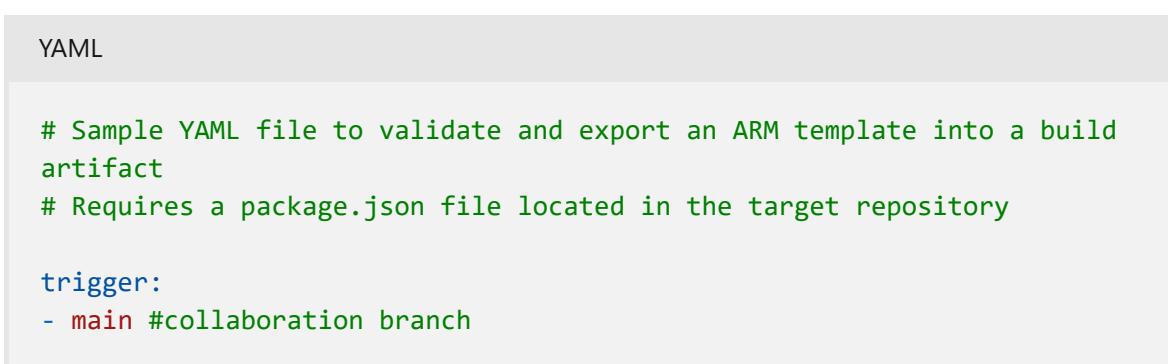


```
JSON
{
  "scripts": {
    "build": "node node_modules/@microsoft/azure-data-factory-utilities/lib/index"
  },
  "dependencies": {
    "@microsoft/azure-data-factory-utilities": "^1.0.0"
  }
}
```

3. Select **Starter pipeline**. If you've uploaded or merged the YAML file, as shown in the following example, you can also point directly at that and edit it.



The screenshot shows the 'Configure your pipeline' blade. At the top, there are tabs: 'Connect', 'Select', 'Configure' (which is highlighted in blue), and 'Review'. Below the tabs, it says 'New pipeline' and 'Configure your pipeline'. There are two main options: 'Starter pipeline' (selected) and 'Existing Azure Pipelines YAML file'. The 'Starter pipeline' option has a description: 'Start with a minimal pipeline that you can customize to build and deploy your code.' The 'Existing Azure Pipelines YAML file' option has a description: 'Select an Azure Pipelines YAML file in any branch of the repository.' At the bottom, there's a 'Show more' button.



```
YAML
# Sample YAML file to validate and export an ARM template into a build artifact
# Requires a package.json file located in the target repository

trigger:
- main #collaboration branch
```

```

pool:
  vmImage: 'ubuntu-latest'

steps:

# Installs Node and the npm packages saved in your package.json file in
# the build

- task: NodeTool@0
  inputs:
    versionSpec: '14.x'
  displayName: 'Install Node.js'

- task: Npm@1
  inputs:
    command: 'install'
    workingDir: '$(Build.Repository.LocalPath)/<folder-of-the-
package.json-file>' #replace with the package.json folder
    verbose: true
  displayName: 'Install npm package'

# Validates all of the Data Factory resources in the repository. You'll
# get the same validation errors as when "Validate All" is selected.
# Enter the appropriate subscription and name for the source factory.
# Either of the "Validate" or "Validate and Generate ARM temmplate"
# options are required to perform validation. Running both is
# unnecessary.

- task: Npm@1
  inputs:
    command: 'custom'
    workingDir: '$(Build.Repository.LocalPath)/<folder-of-the-
package.json-file>' #replace with the package.json folder
    customCommand: 'run build validate
$(Build.Repository.LocalPath)/<Root-folder-from-Git-configuration-
settings-in-ADF> /subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxx/resourceGroups/<Your-ResourceGroup-
Name>/providers/Microsoft.DataFactory/factories/<Your-Factory-Name>'
  displayName: 'Validate'

# Validate and then generate the ARM template into the destination
# folder, which is the same as selecting "Publish" from the UX.
# The ARM template generated isn't published to the live version of the
# factory. Deployment should be done by using a CI/CD pipeline.

- task: Npm@1
  inputs:
    command: 'custom'
    workingDir: '$(Build.Repository.LocalPath)/<folder-of-the-
package.json-file>' #replace with the package.json folder
    customCommand: 'run build export
$(Build.Repository.LocalPath)/<Root-folder-from-Git-configuration-
settings-in-ADF> /subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxx/resourceGroups/<Your-ResourceGroup-
```

```

Name>/providers/Microsoft.DataFactory/factories/<Your-Factory-Name>
"ArmTemplate"
#For using preview that allows you to only stop/ start triggers that
are modified, please comment out the above line and uncomment the below
line. Make sure the package.json contains the build-preview command.
  #customCommand: 'run build-preview export
$(Build.Repository.LocalPath) /subscriptions/222f1459-6ebd-4896-82ab-
652d5f6883cf/resourceGroups/GartnerMQ2021/providers/Microsoft.DataFacto
ry/factories/Dev-GartnerMQ2021-DataFactory "ArmTemplate"'
  displayName: 'Validate and Generate ARM template'

# Publish the artifact to be used as a source for a release pipeline.

- task: PublishPipelineArtifact@1
  inputs:
    targetPath: '$(Build.Repository.LocalPath)/<folder-of-the-
package.json-file>/ArmTemplate' #replace with the package.json folder
    artifact: 'ArmTemplates'
    publishLocation: 'pipeline'

```

ⓘ Note

Node version 10.x is currently still supported but may be deprecated in the future. We highly recommend upgrading to the latest version.

4. Enter your YAML code. We recommend that you use the YAML file as a starting point.
5. Save and run. If you used the YAML, it gets triggered every time the main branch is updated.

ⓘ Note

The generated artifacts already contain pre and post deployment scripts for the triggers so it isn't necessary to add these manually. However, when deploying one would still need to reference the [documentation on stopping and starting triggers](#) to execute the provided script.

Related content

Learn more information about continuous integration and delivery in Data Factory: [Continuous integration and delivery in Azure Data Factory](#).

Git integration with Databricks Repos

Article • 01/23/2024

Databricks Repos is a visual Git client and API in Azure Databricks. It supports common Git operations such as cloning a repository, committing and pushing, pulling, branch management, and visual comparison of diffs when committing.

Within Repos you can develop code in notebooks or other files and follow data science and engineering code development best practices using Git for version control, collaboration, and CI/CD.

What can you do with Databricks Repos?

Databricks Repos provides source control for data and AI projects by integrating with Git providers.

In Databricks Repos, you can use Git functionality to:

- Clone, push to, and pull from a remote Git repository.
- Create and manage branches for development work, including merging, rebasing, and resolving conflicts.
- Create notebooks (including IPYNB notebooks) and edit them and other files.
- Visually compare differences upon commit and resolve merge conflicts.

For step-by-step instructions, see [Run Git operations on Databricks Repos](#).

ⓘ Note

Databricks Repos also has an [API](#) that you can integrate with your CI/CD pipeline. For example, you can programmatically update a Databricks repo so that it always has the most recent version of the code. For information about best practices for code development using Databricks Repos, see [CI/CD techniques with Git and Databricks Repos](#).

For information on the kinds of notebooks supported in Azure Databricks, see [Export and import Databricks notebooks](#).

Supported Git providers

Databricks Git folders are backed by an integrated Git repository. The repository can be hosted by any of the cloud and enterprise Git providers listed in the following section.

 **Note**

What is a “Git provider”?

A “Git provider” is the specific (named) service that hosts a source control model based on Git. Git-based source control platforms are hosted in two ways: as a cloud service hosted by the developing company, or as an on-premises service installed and managed by your own company on its own hardware. Many Git providers such as GitHub, Microsoft, GitLab, and Atlassian provide both cloud-based SaaS and on-premises (sometimes called “self-managed”) Git services.

When choosing your Git provider during configuration, you must be aware of the differences between cloud (SaaS) and on-premises Git providers. On-premises solutions are typically hosted behind a company VPN and might not be accessible from the internet. Usually, the on-premises Git providers have a name ending in “Server” or “Self-Managed”, but if you are uncertain, contact your company admins or review the Git provider’s documentation.

 **Note**

If you are using “GitHub” as a provider and are still uncertain if you are using the cloud or on-premises version, see [About GitHub Enterprise Server](#) in the GitHub docs.

Cloud Git providers supported by Databricks

- GitHub, GitHub AE, and GitHub Enterprise Cloud
- Atlassian BitBucket Cloud
- GitLab and GitLab EE
- Microsoft Azure DevOps (Azure Repos)

On-premises Git providers supported by Databricks

- GitHub Enterprise Server
- Atlassian BitBucket Server and Data Center
- GitLab Self-Managed

- Microsoft Azure DevOps Server: A workspace admin must explicitly allowlist the URL domain prefixes for your Microsoft Azure DevOps Server if the URL does not match `dev.azure.com/*` or `visualstudio.com/*`. For more details, see [Restrict usage to URLs in an allow list](#)

If you are integrating an on-premises Git repo that is not accessible from the internet, a proxy for Git authentication requests must also be installed within your company's VPN. For more details, see [Set up private Git connectivity for Databricks Repos](#).

To learn how to use access tokens with your Git provider, see [Configure Git credentials & connect a remote repo to Azure Databricks](#).

Resources for Git integration

Use the Databricks CLI 2.0 for Git integration with Azure Databricks:

- [Download the latest CLI version ↗](#)
- [Set up the CLI](#)

Read the following reference docs:

- [Databricks CLI global flags and commands](#)
- [Databricks CLI reference ↗](#)

Next steps

- [Set up Databricks Repos](#)
- [Configure Git credentials & connect a remote repo to Azure Databricks](#)

Git version control for notebooks (legacy)

Article • 01/15/2024

ⓘ Important

Legacy notebook Git integration support will be removed on January 31st, 2024.

Databricks recommends that you use Databricks Repos to sync your work in Databricks with a remote Git repository.

This article describes how to set up Git version control for notebooks (legacy feature). You can also use the [Databricks CLI](#) or [Workspace API](#) to import and export notebooks and to perform Git operations in your local development environment.

Enable and disable Git versioning

By default version control is enabled. To toggle this setting:

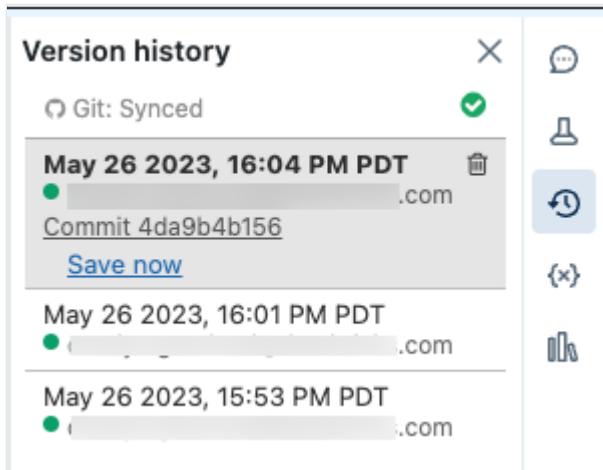
1. Go to **Admin settings** > **Workspace settings**.
2. In the **Advanced** section, deselect the **Notebook Git Versioning** toggle.

Configure version control

To configure version control, create access credentials in your Git provider, then add those credentials to Azure Databricks.

Work with notebook versions

You work with notebook versions in the history panel. Open the history panel by clicking  in the right sidebar.

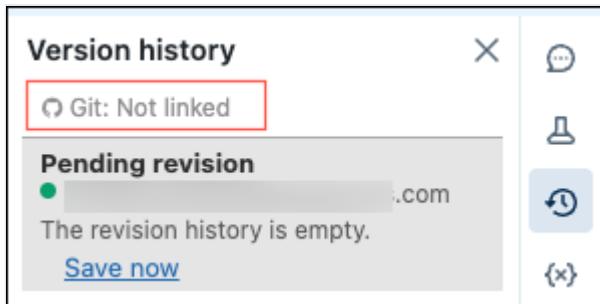


ⓘ Note

You cannot modify a notebook while the history panel is open.

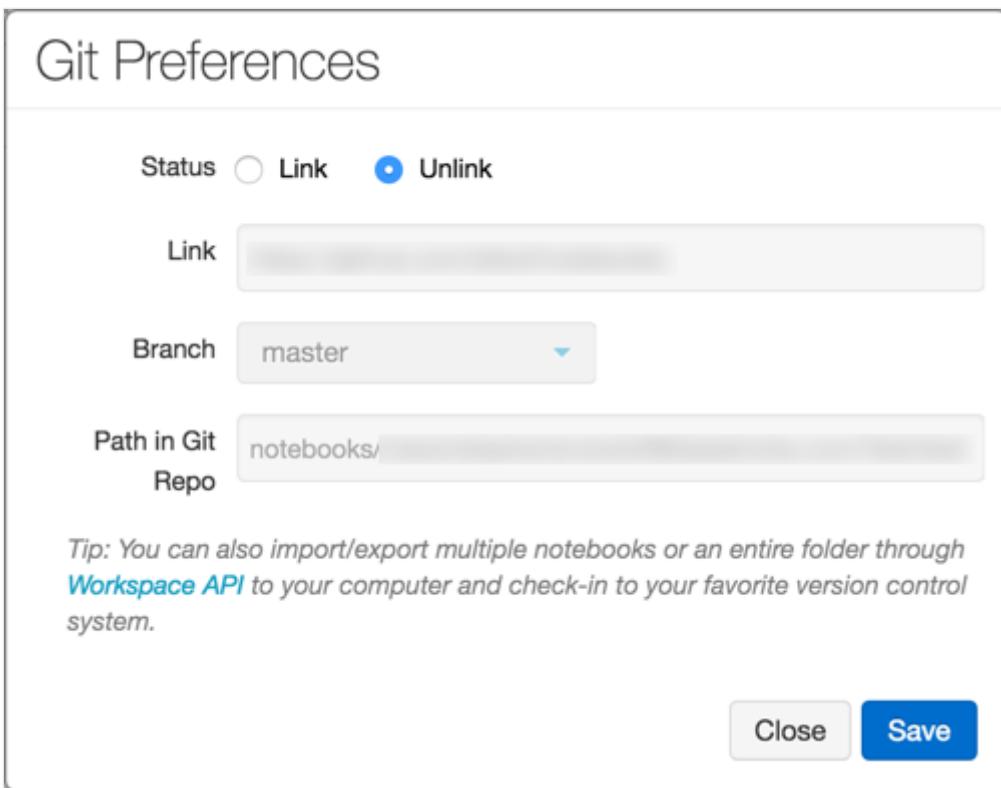
Link a notebook to GitHub

1. Click  in the right sidebar. The Git status bar displays **Git: Not linked**.



2. Click **Git: Not linked**.

The Git Preferences dialog appears. The first time you open your notebook, the Status is **Unlink**, because the notebook is not in GitHub.

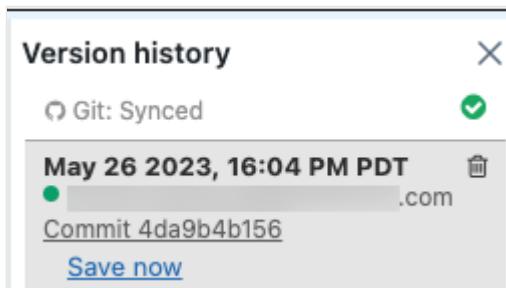


3. In the Status field, click **Link**.
4. In the Link field, paste the URL of the GitHub repository.
5. Click the **Branch** drop-down and select a branch or type the name of a new branch.
6. In the Path in Git Repo field, specify where in the repository to store your file.
Python notebooks have the suggested default file extension `.py`. If you use `.ipynb`, your notebook will save in iPython notebook format. If the file already exists on GitHub, you can directly copy and paste the URL of the file.
7. Click **Save** to finish linking your notebook. If this file did not previously exist, a prompt with the option **Save this file to your GitHub repo** displays.
8. Type a message and click **Save**.

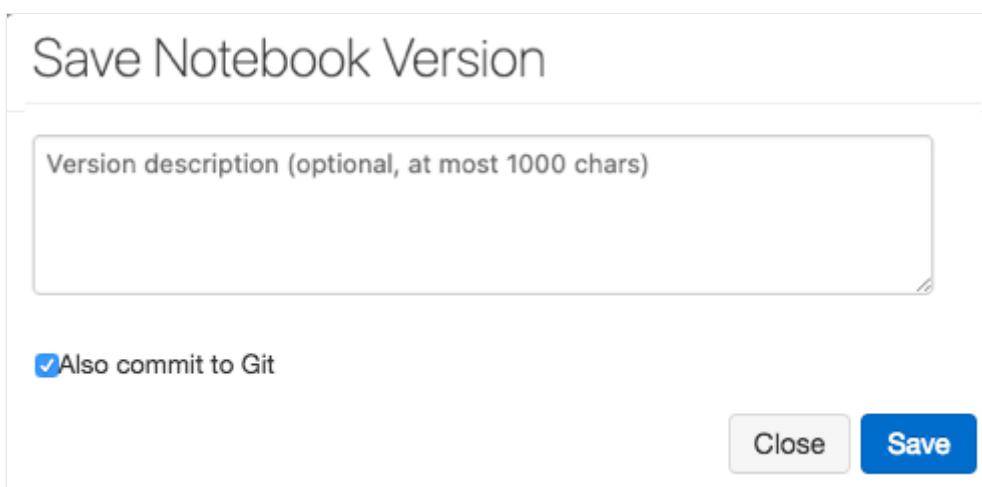
Save a notebook to GitHub

While the changes that you make to your notebook are saved automatically to the Azure Databricks version history, changes do not automatically persist to GitHub.

1. Click  in the right sidebar to open the history panel.



2. Click **Save Now** to save your notebook to GitHub. The Save Notebook Version dialog appears.
3. Optionally, enter a message to describe your change.
4. Make sure that **Also commit to Git** is selected.

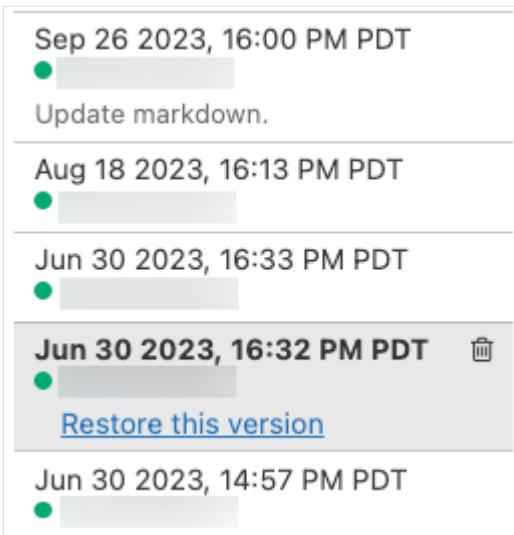


5. Click **Save**.

Revert or update a notebook to a version from GitHub

Once you link a notebook, Azure Databricks syncs your history with Git every time you re-open the history panel. Versions that sync to Git have commit hashes as part of the entry.

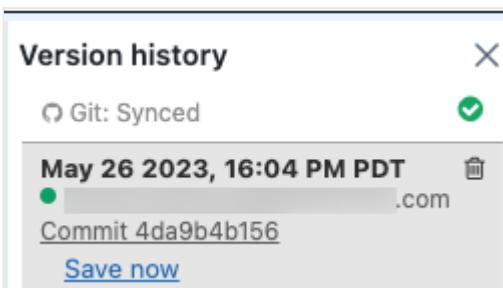
1. Click  in the right sidebar to open the history panel.



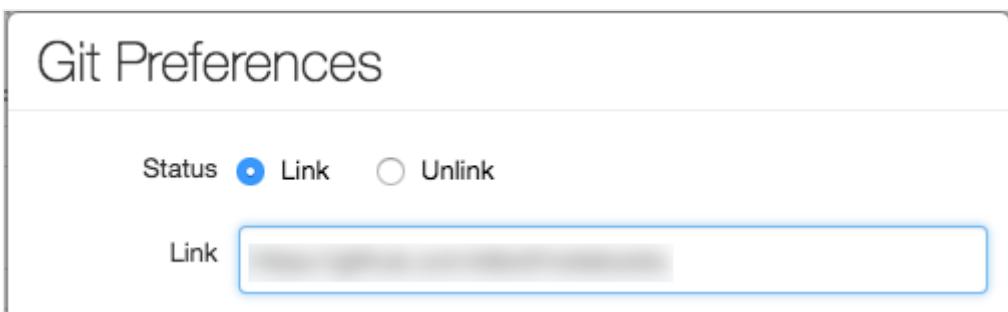
2. Choose an entry in the history panel. Azure Databricks displays that version.
3. Click **Restore this version**.
4. Click **Confirm** to confirm that you want to restore that version.

Unlink a notebook

1. Click  in the right sidebar to open the history panel.
2. The Git status bar displays **Git: Synced**.



3. Click **Git: Synced**.



4. In the Git Preferences dialog, click **Unlink**.
5. Click **Save**.

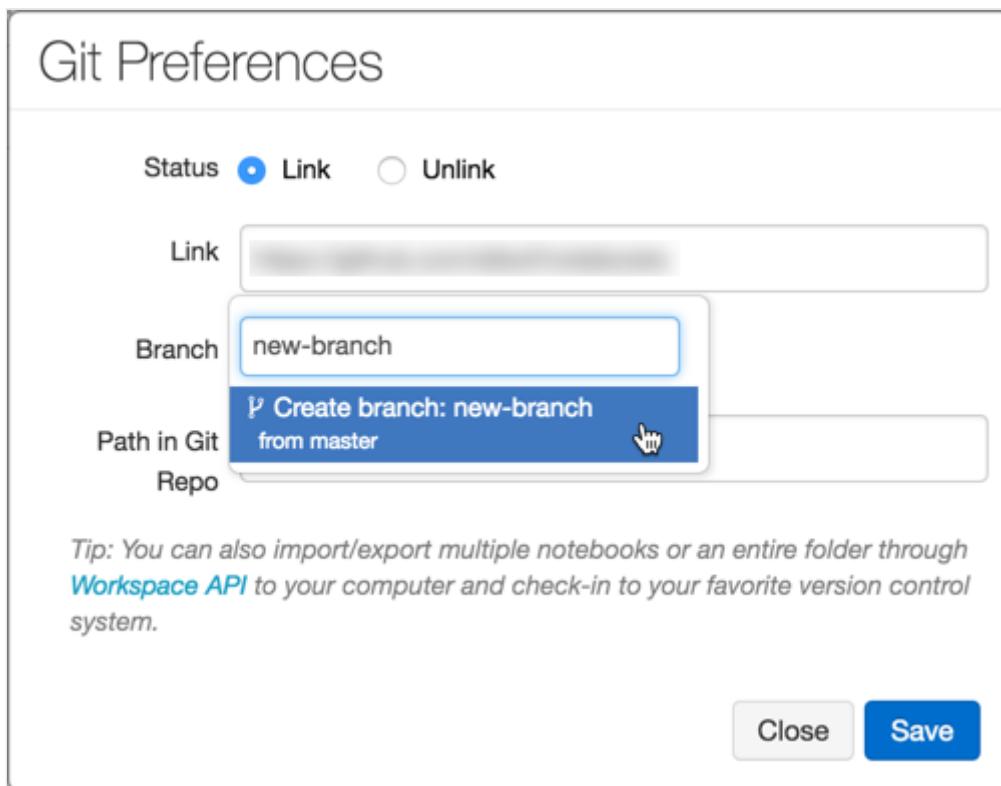
6. Click **Confirm** to confirm that you want to unlink the notebook from version control.

Use branches

You can work on any branch of your repository and create new branches inside Azure Databricks.

Create a branch

1. Click  in the right sidebar to open the history panel.
2. Click the Git status bar to open the GitHub panel.
3. Click the **Branch** dropdown.
4. Enter a branch name.

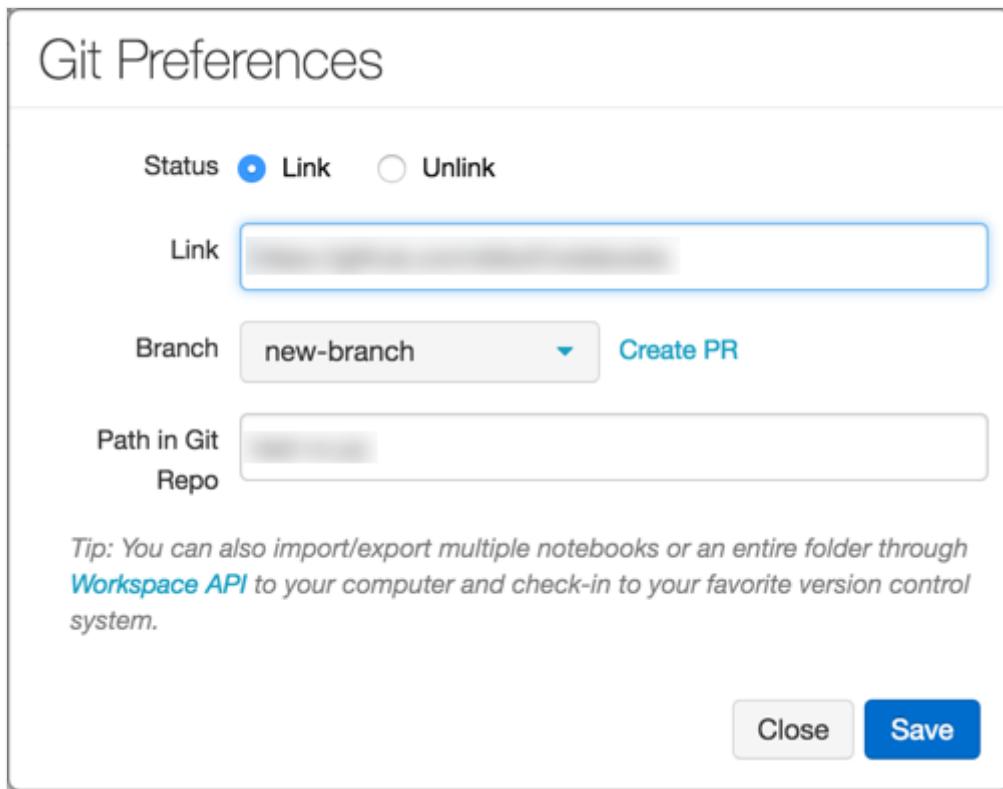


5. Select the **Create Branch** option at the bottom of the dropdown. The parent branch is indicated. You always branch from your current selected branch.

Create a pull request

1. Click  in the right sidebar to open the history panel.

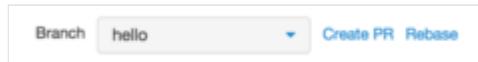
2. Click the Git status bar to open the GitHub panel.



3. Click Create PR. GitHub opens to a pull request page for the branch.

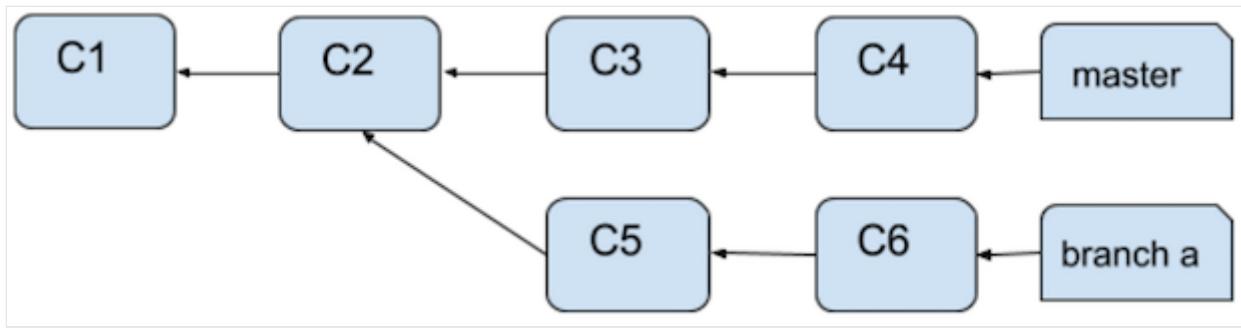
Rebase a branch

You can also rebase your branch inside Azure Databricks. The **Rebase** link displays if new commits are available in the parent branch. Only rebasing on top of the default branch of the parent repository is supported.

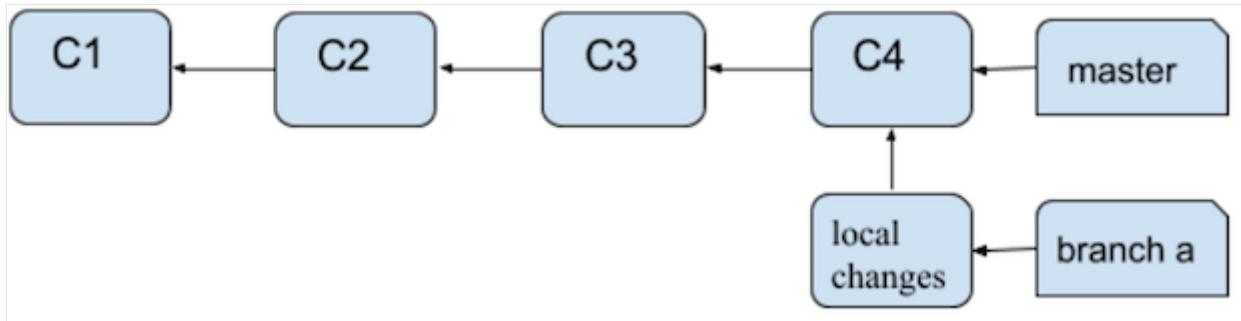


For example, assume that you are working on `databricks/reference-apps`. You fork it into your own account (for example, `brkyvz`) and start working on a branch called `my-branch`. If a new update is pushed to `databricks:master`, then the `Rebase` button displays, and you will be able to pull the changes into your branch `brkyvz:my-branch`.

Rebasing works a little differently in Azure Databricks. Assume the following branch structure:



After a rebase, the branch structure looks like:



What's different here is that Commits C5 and C6 do not apply on top of C4. They appear as local changes in your notebook. Merge conflicts show up as follows:

```

> 3 + 3

> <<<< 4f0ebd02fb668f7ecea5949a543c298620939ce6
"Hello World!"
=====
"Hello Universe!"
>>>> HEAD
  
```

You can then commit to GitHub once again using the **Save Now** button.

What happens if someone branched off from my branch that I just rebased?

If your branch (for example, `branch-a`) was the base for another branch (`branch-b`), and you rebase, you need not worry! Once a user also rebases `branch-b`, everything will work out. The best practice in this situation is to use separate branches for separate notebooks.

Best practices for code reviews

Azure Databricks supports Git branching.

- You can link a notebook to any branch in a repository. Azure Databricks recommends using a separate branch for each notebook.
- During development, you can link a notebook to a fork of a repository or to a non-default branch in the main repository. To integrate your changes upstream, you can use the **Create PR** link in the **Git Preferences** dialog in Azure Databricks to create a GitHub pull request. The Create PR link displays only if you're not working on the default branch of the parent repository.

Troubleshooting

If you receive errors related to syncing GitHub history, verify the following:

- You can only link a notebook to an initialized Git repository that isn't empty. Test the URL in a web browser.
- The GitHub personal access token must be active.
- To use a private GitHub repository, you must have permission to read the repository.
- If a notebook is linked to a GitHub branch that is renamed, the change is not automatically reflected in Azure Databricks. You must re-link the notebook to the branch manually.

Migrate to Databricks Repos

Users that need to migrate to Databricks Repos from the legacy Git version control can use the following guide:

- [Switching to Databricks Repos from Legacy Git integration ↗](#)

Configure Git credentials & connect a remote repo to Azure Databricks

Article • 01/26/2024

This article describes how to configure your Git credentials in Databricks so that you can connect a remote repo to Databricks Repos.

For a list of supported Git providers (cloud and on-premises), read [Supported Git providers](#).

Configuring GitHub and GitHub AE credentials

The following information applies to GitHub and GitHub AE users.

Why use the Databricks GitHub App instead of a PAT?

Databricks Repos allows you to choose the Databricks GitHub App for user authentication instead of PATs if you are using a hosted GitHub account. Using the GitHub App provides the following benefits over PATs:

- It uses OAuth 2.0 for user authentication. OAuth 2.0 repo traffic is encrypted for strong security.
- It is easier to integrate ([see the steps below](#)) and does not require individual tracking of tokens.
- Token renewal is handled automatically.
- The integration can be scoped to specific attached Git repos, allowing you more granular control over access.

Important

As per standard OAuth 2.0 integration, Databricks stores a user's access and refresh tokens—all other access control is handled by GitHub. Access and refresh tokens follow GitHub's default expiry rules with access tokens expiring after 8 hours (which minimizes risk in the event of credential leak). Refresh tokens have a 6-month lifetime if unused. Linked credentials expire after 6 months of inactivity, requiring the user to reconfigure them.

You can optionally encrypt Databricks tokens using [customer-managed keys](#) (CMK).

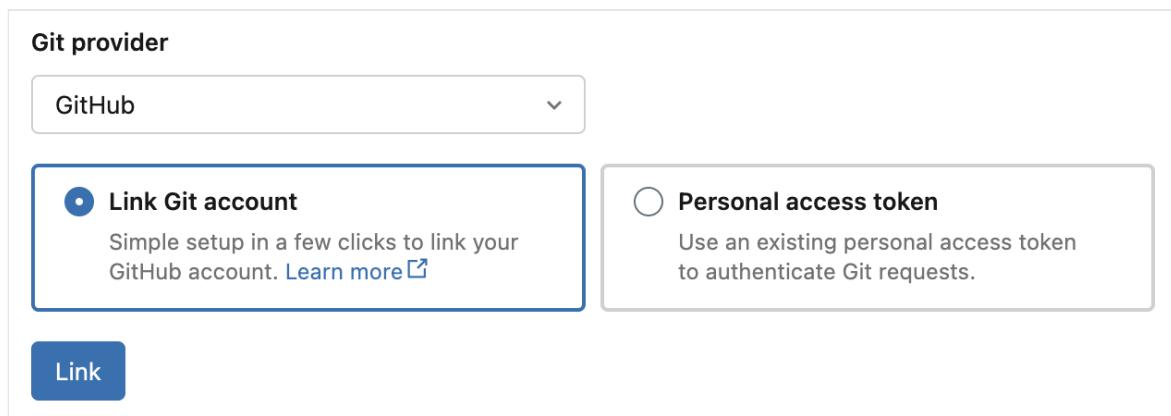
Link GitHub account using Databricks GitHub App

Note

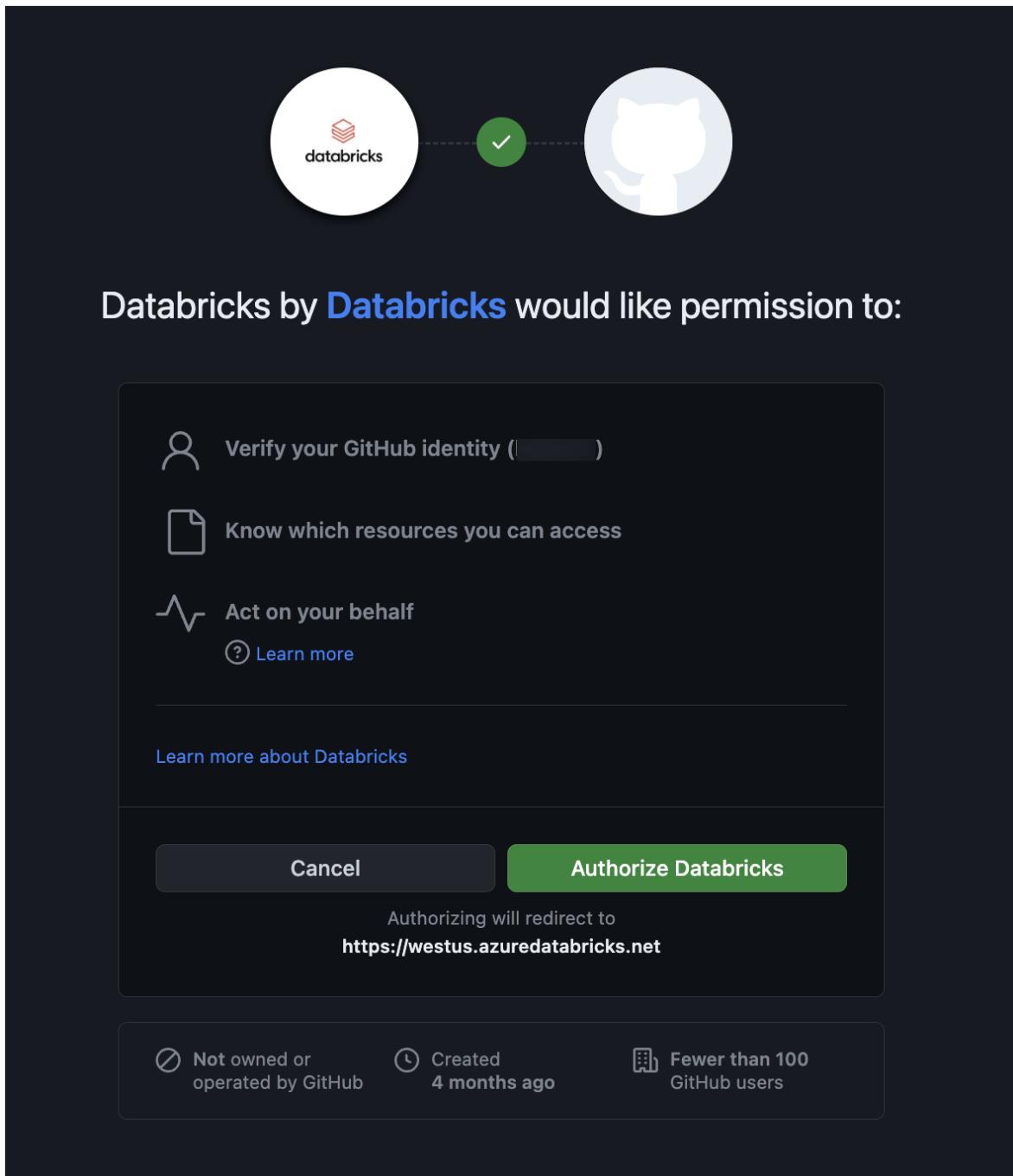
- This feature is not supported in GitHub Enterprise Server. Use a personal access token instead.

In Azure Databricks, link your GitHub account on the User Settings page:

1. In the upper-right corner of any page, click your username, then select **User Settings**.
2. Click the **Linked accounts** tab.
3. Change your provider to GitHub, select **Link Git account**, and click **Link**.



4. The Databricks GitHub app authorization page appears. Authorize the app to complete the setup. Authorizing the app allows Databricks to act on your behalf when you perform Git operations in Repos (such as cloning a repository). See the [GitHub documentation](#) for more details on app authorization.

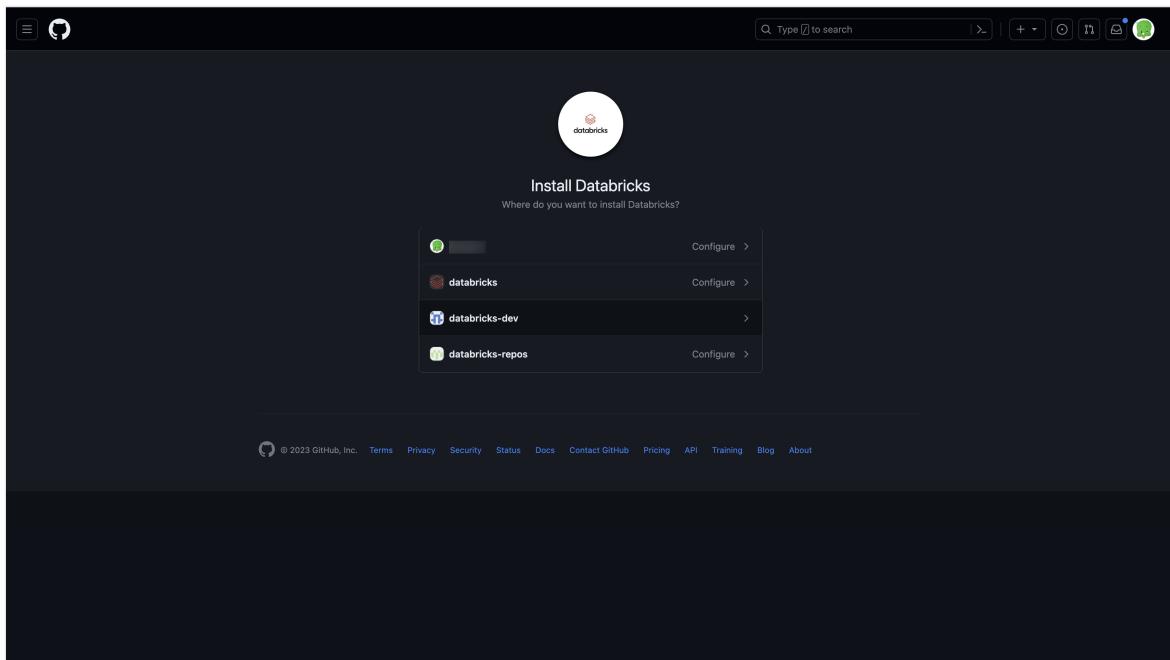


5. To allow access to GitHub repositories, follow the steps below to install and configure the Databricks GitHub app.

Install and configure the [Databricks GitHub app](#) to allow access to repositories

You must install and configure the Databricks GitHub app on GitHub repositories that you want to access from Databricks Repos. See the [GitHub documentation](#) for more details on app installation.

1. Open the [Databricks GitHub app installation page](#).
2. Select the account that owns the repositories you want to access.



3. If you are not an owner of the account, you must have the account owner install and configure the app for you.
4. If you are the account owner, install the app. Installing the app gives read and write access to code. Code is only accessed on behalf of users (for example, when a user clones a repository in Databricks Repos).
5. Optionally, you can give access to only a subset of repositories by selecting the **Only select repositories** option.

Connect to a GitHub repo using a personal access token

In GitHub, follow these steps to create a personal access token that allows access to your repositories:

1. In the upper-right corner of any page, click your profile photo, then click **Settings**.
2. Click **Developer settings**.
3. Click the **Personal access tokens** tab.
4. Click the **Generate new token** button.
5. Enter a token description.
6. Select the **repo** scope and **workflow** scope, and click the **Generate token** button. **workflow** scope is needed in case your repository has GitHub Action workflows.

OAuth Apps

GitHub Apps

Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Token description

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations

7. Copy the token to your clipboard. You enter this token in Azure Databricks under [User Settings > Linked accounts](#).

To use single sign-on, see [Authorizing a personal access token for use with SAML single sign-on](#).

GitLab

In GitLab, follow these steps to create a personal access token that allows access to your repositories:

1. From GitLab, click your user icon in the upper right corner of the screen and select [Preferences](#).
2. Click [Access Tokens](#) in the sidebar.

GitLab Projects Groups More Search or jump to... User Settings > Access Tokens

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Name

Expires at

Scopes

- api**
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.
- read_user**
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- read_api**
Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- read_repository**
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- write_repository**
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- read_registry**
Grants read-only access to container registry images on private projects.
- write_registry**
Grants write access to container registry images on private projects.

Create personal access token

3. Enter a name for the token.
4. Check the `read_repository` and `write_repository` permissions, and click **Create personal access token**.
5. Copy the token to your clipboard. Enter this token in Azure Databricks under **User Settings > Linked accounts**.

See the [GitLab documentation](#) to learn more about how to create and manage personal access tokens.

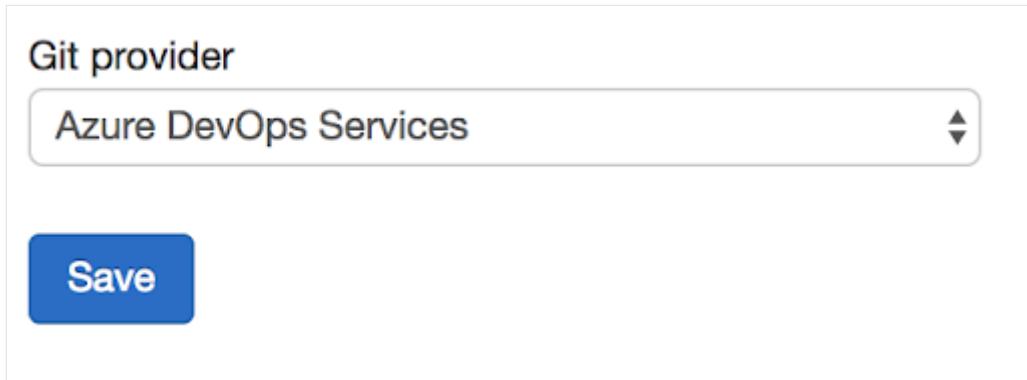
Azure DevOps Services

Connect to an Azure DevOps repo using Microsoft Entra ID (formerly Azure Active Directory)

Authentication with [Azure DevOps Services](#) is done automatically when you authenticate using Microsoft Entra ID. The Azure DevOps Services organization must be linked to the same Microsoft Entra ID tenant as Databricks.

In Azure Databricks, set your Git provider to Azure DevOps Services on the User Settings page:

1. In the upper-right corner of any page, click your username, then select **User Settings**.
2. Click the **Linked accounts** tab.
3. Change your provider to Azure DevOps Services.



Connect to an Azure DevOps repo using a token

The following steps show you how to connect an Azure Databricks repo to an Azure DevOps repo when they aren't in the same Microsoft Entra ID tenancy.

Get an access token for the repository in Azure DevOps:

1. Go to dev.azure.com, and then sign in to the DevOps organization containing the repository you want to connect Azure Databricks to.
2. In the upper-right side, click the User Settings icon and select **Personal Access Tokens**.
3. Click **+ New Token**.
4. Enter information into the form:
 - a. Name the token.
 - b. Select the organization name, which is the repo name.
 - c. Set an expiration date.
 - d. Choose the the scope required, such as **Full access**.
5. Copy the access token displayed.
6. Enter this token in Azure Databricks under **User Settings > Linked accounts**.
7. In **Git provider username or email**, enter the email address you use to log in to the DevOps organization.

Bitbucket

In Bitbucket, follow these steps to create an app password that allows access to your repositories:

1. Go to Bitbucket Cloud and create an app password that allows access to your repositories. See the [Bitbucket Cloud documentation](#).
2. Record the password.
3. In Azure Databricks, enter this password under **User Settings > Linked accounts**.

Configure Git credentials & connect a remote repo to Azure Databricks

Article • 01/26/2024

This article describes how to configure your Git credentials in Databricks so that you can connect a remote repo to Databricks Repos.

For a list of supported Git providers (cloud and on-premises), read [Supported Git providers](#).

Configuring GitHub and GitHub AE credentials

The following information applies to GitHub and GitHub AE users.

Why use the Databricks GitHub App instead of a PAT?

Databricks Repos allows you to choose the Databricks GitHub App for user authentication instead of PATs if you are using a hosted GitHub account. Using the GitHub App provides the following benefits over PATs:

- It uses OAuth 2.0 for user authentication. OAuth 2.0 repo traffic is encrypted for strong security.
- It is easier to integrate ([see the steps below](#)) and does not require individual tracking of tokens.
- Token renewal is handled automatically.
- The integration can be scoped to specific attached Git repos, allowing you more granular control over access.

Important

As per standard OAuth 2.0 integration, Databricks stores a user's access and refresh tokens—all other access control is handled by GitHub. Access and refresh tokens follow GitHub's default expiry rules with access tokens expiring after 8 hours (which minimizes risk in the event of credential leak). Refresh tokens have a 6-month lifetime if unused. Linked credentials expire after 6 months of inactivity, requiring the user to reconfigure them.

You can optionally encrypt Databricks tokens using [customer-managed keys](#) (CMK).

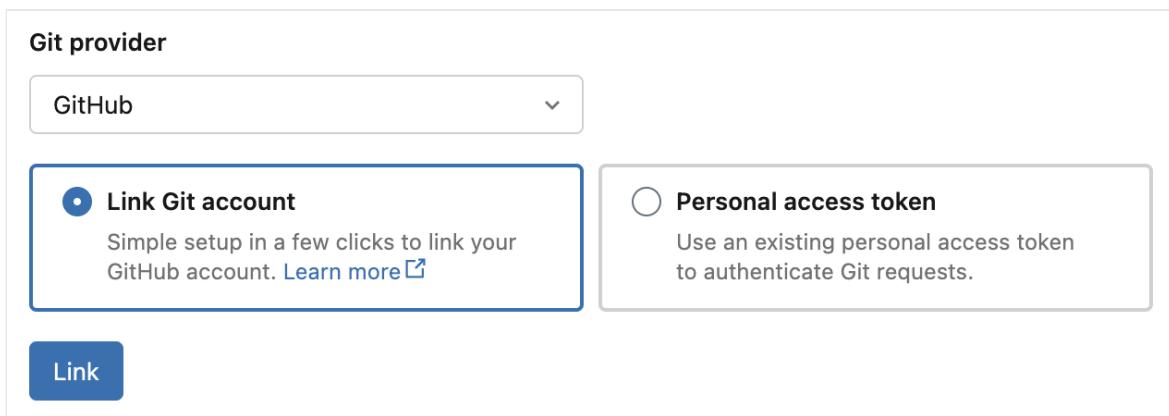
Link GitHub account using Databricks GitHub App

Note

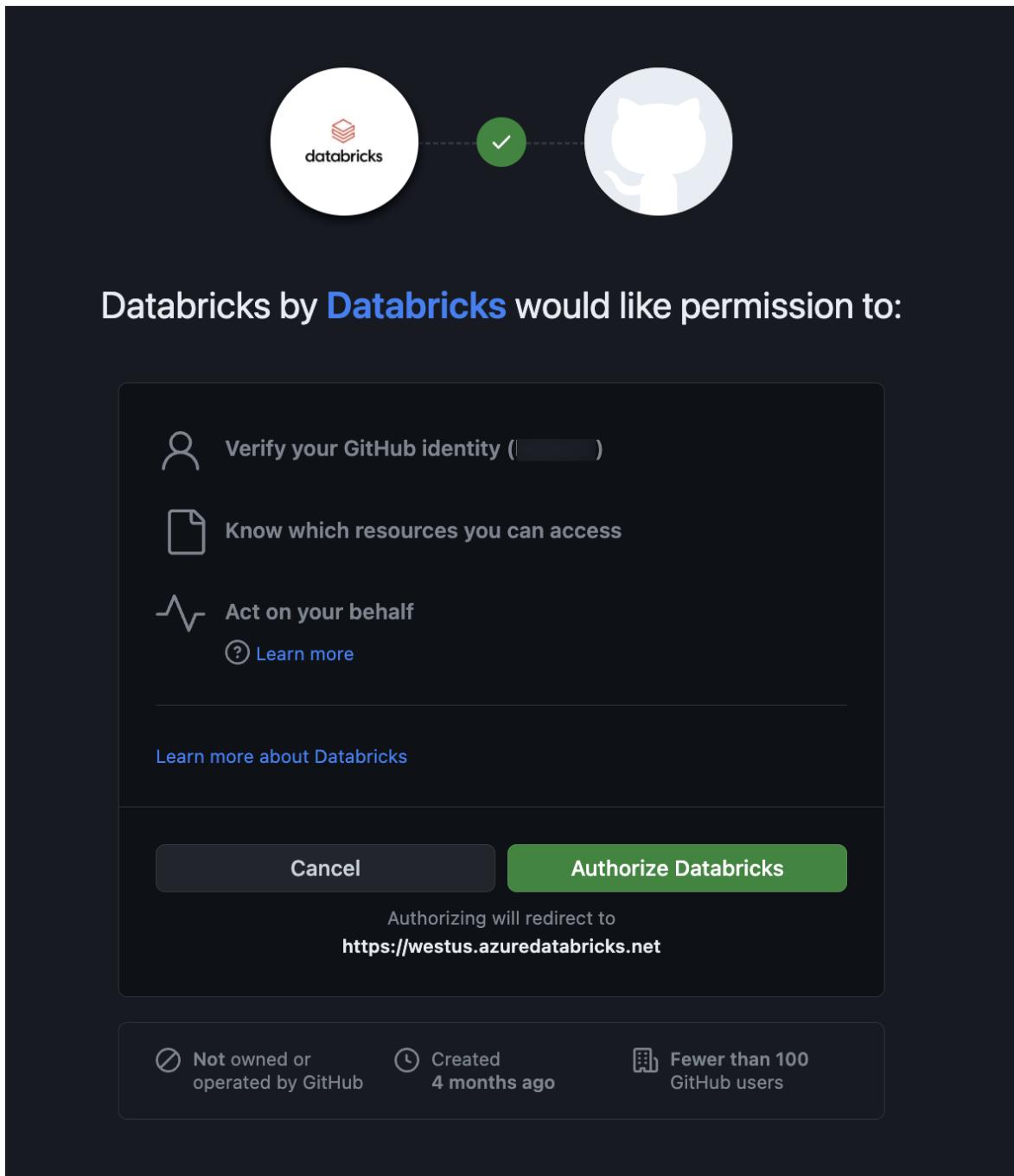
- This feature is not supported in GitHub Enterprise Server. Use a personal access token instead.

In Azure Databricks, link your GitHub account on the User Settings page:

1. In the upper-right corner of any page, click your username, then select **User Settings**.
2. Click the **Linked accounts** tab.
3. Change your provider to GitHub, select **Link Git account**, and click **Link**.



4. The Databricks GitHub app authorization page appears. Authorize the app to complete the setup. Authorizing the app allows Databricks to act on your behalf when you perform Git operations in Repos (such as cloning a repository). See the [GitHub documentation](#) for more details on app authorization.

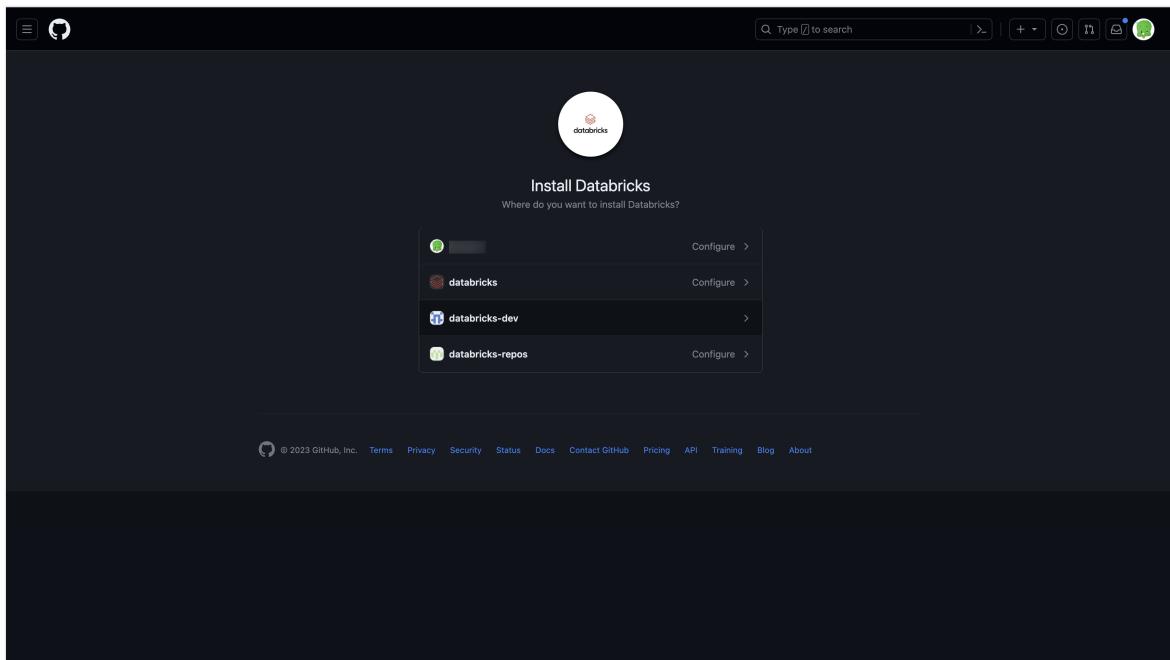


5. To allow access to GitHub repositories, follow the steps below to install and configure the Databricks GitHub app.

Install and configure the [Databricks GitHub app](#) to allow access to repositories

You must install and configure the Databricks GitHub app on GitHub repositories that you want to access from Databricks Repos. See the [GitHub documentation](#) for more details on app installation.

1. Open the [Databricks GitHub app installation page](#).
2. Select the account that owns the repositories you want to access.



3. If you are not an owner of the account, you must have the account owner install and configure the app for you.
4. If you are the account owner, install the app. Installing the app gives read and write access to code. Code is only accessed on behalf of users (for example, when a user clones a repository in Databricks Repos).
5. Optionally, you can give access to only a subset of repositories by selecting the **Only select repositories** option.

Connect to a GitHub repo using a personal access token

In GitHub, follow these steps to create a personal access token that allows access to your repositories:

1. In the upper-right corner of any page, click your profile photo, then click **Settings**.
2. Click **Developer settings**.
3. Click the **Personal access tokens** tab.
4. Click the **Generate new token** button.
5. Enter a token description.
6. Select the **repo** scope and **workflow** scope, and click the **Generate token** button. **workflow** scope is needed in case your repository has GitHub Action workflows.

OAuth Apps

GitHub Apps

Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Token description

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations

7. Copy the token to your clipboard. You enter this token in Azure Databricks under [User Settings > Linked accounts](#).

To use single sign-on, see [Authorizing a personal access token for use with SAML single sign-on](#).

GitLab

In GitLab, follow these steps to create a personal access token that allows access to your repositories:

1. From GitLab, click your user icon in the upper right corner of the screen and select [Preferences](#).
2. Click [Access Tokens](#) in the sidebar.

GitLab Projects Groups More Search or jump to... User Settings > Access Tokens

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Name

Expires at

Scopes

- api**
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.
- read_user**
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- read_api**
Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- read_repository**
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- write_repository**
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- read_registry**
Grants read-only access to container registry images on private projects.
- write_registry**
Grants write access to container registry images on private projects.

Create personal access token

3. Enter a name for the token.
4. Check the `read_repository` and `write_repository` permissions, and click **Create personal access token**.
5. Copy the token to your clipboard. Enter this token in Azure Databricks under **User Settings > Linked accounts**.

See the [GitLab documentation](#) to learn more about how to create and manage personal access tokens.

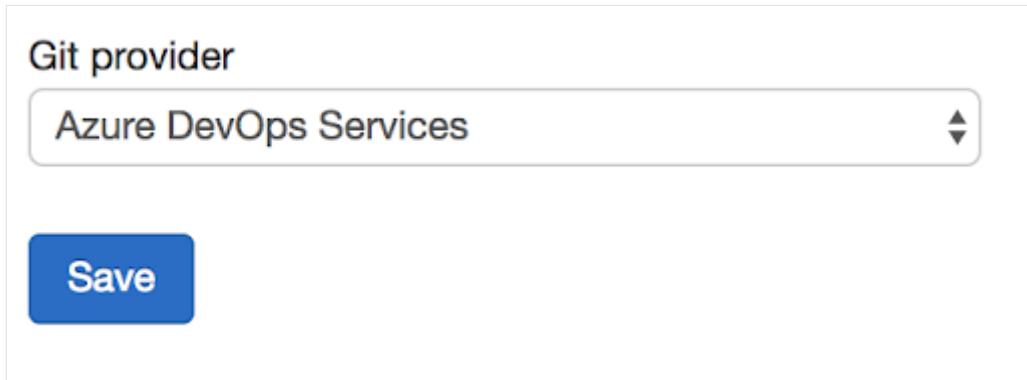
Azure DevOps Services

Connect to an Azure DevOps repo using Microsoft Entra ID (formerly Azure Active Directory)

Authentication with [Azure DevOps Services](#) is done automatically when you authenticate using Microsoft Entra ID. The Azure DevOps Services organization must be linked to the same Microsoft Entra ID tenant as Databricks.

In Azure Databricks, set your Git provider to Azure DevOps Services on the User Settings page:

1. In the upper-right corner of any page, click your username, then select **User Settings**.
2. Click the **Linked accounts** tab.
3. Change your provider to Azure DevOps Services.



Connect to an Azure DevOps repo using a token

The following steps show you how to connect an Azure Databricks repo to an Azure DevOps repo when they aren't in the same Microsoft Entra ID tenancy.

Get an access token for the repository in Azure DevOps:

1. Go to dev.azure.com, and then sign in to the DevOps organization containing the repository you want to connect Azure Databricks to.
2. In the upper-right side, click the User Settings icon and select **Personal Access Tokens**.
3. Click **+ New Token**.
4. Enter information into the form:
 - a. Name the token.
 - b. Select the organization name, which is the repo name.
 - c. Set an expiration date.
 - d. Choose the the scope required, such as **Full access**.
5. Copy the access token displayed.
6. Enter this token in Azure Databricks under **User Settings > Linked accounts**.
7. In **Git provider username or email**, enter the email address you use to log in to the DevOps organization.

Bitbucket

In Bitbucket, follow these steps to create an app password that allows access to your repositories:

1. Go to Bitbucket Cloud and create an app password that allows access to your repositories. See the [Bitbucket Cloud documentation](#).
2. Record the password.
3. In Azure Databricks, enter this password under **User Settings > Linked accounts**.

Git version control for notebooks (legacy)

Article • 01/15/2024

ⓘ Important

Legacy notebook Git integration support will be removed on January 31st, 2024.

Databricks recommends that you use Databricks Repos to sync your work in Databricks with a remote Git repository.

This article describes how to set up Git version control for notebooks (legacy feature). You can also use the [Databricks CLI](#) or [Workspace API](#) to import and export notebooks and to perform Git operations in your local development environment.

Enable and disable Git versioning

By default version control is enabled. To toggle this setting:

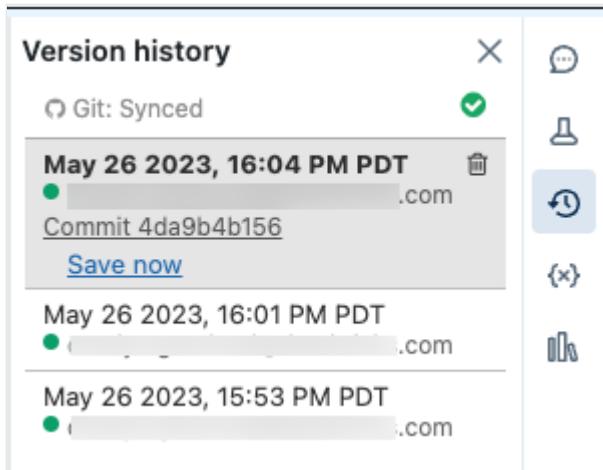
1. Go to **Admin settings** > **Workspace settings**.
2. In the **Advanced** section, deselect the **Notebook Git Versioning** toggle.

Configure version control

To configure version control, create access credentials in your Git provider, then add those credentials to Azure Databricks.

Work with notebook versions

You work with notebook versions in the history panel. Open the history panel by clicking  in the right sidebar.

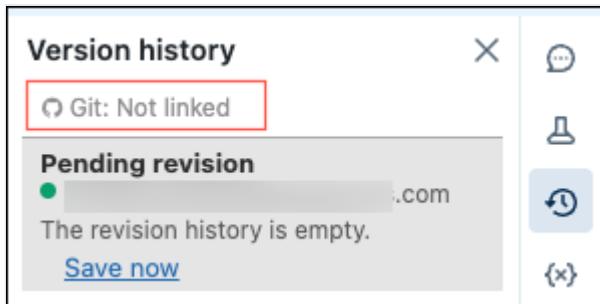


ⓘ Note

You cannot modify a notebook while the history panel is open.

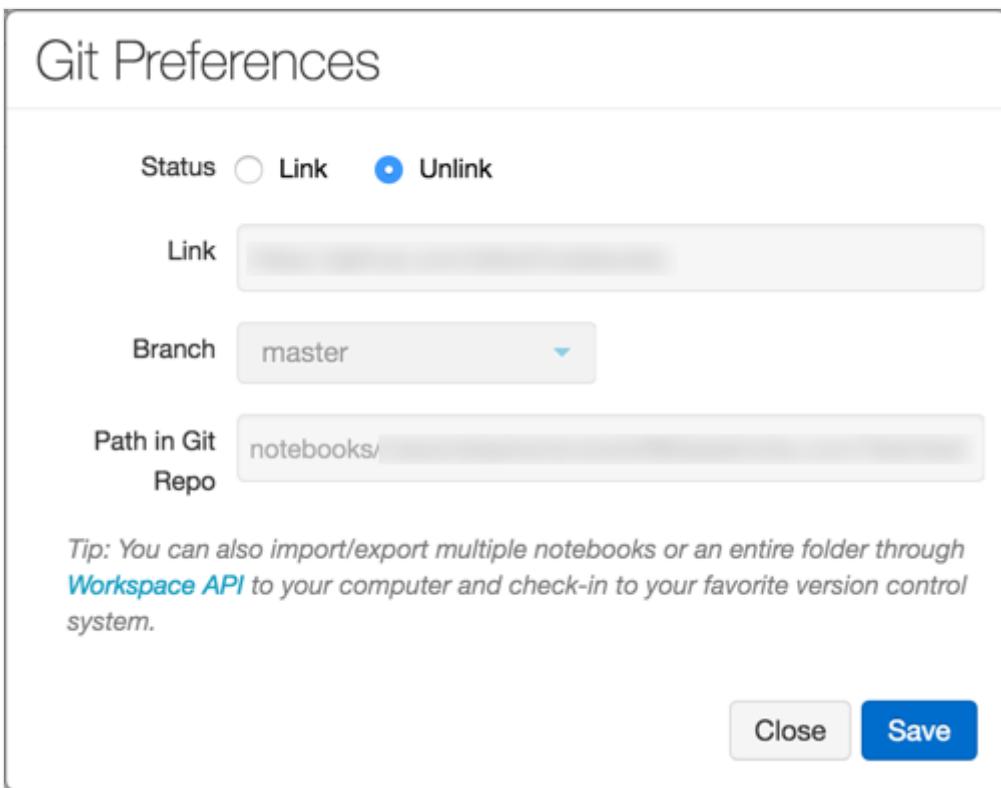
Link a notebook to GitHub

1. Click  in the right sidebar. The Git status bar displays **Git: Not linked**.



2. Click **Git: Not linked**.

The Git Preferences dialog appears. The first time you open your notebook, the Status is **Unlink**, because the notebook is not in GitHub.

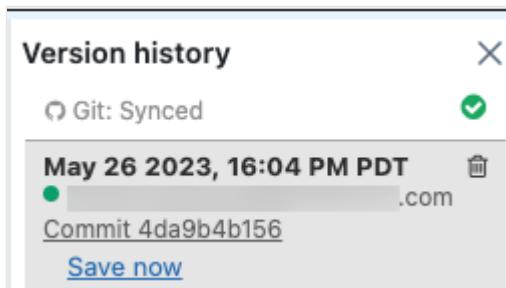


3. In the Status field, click **Link**.
4. In the Link field, paste the URL of the GitHub repository.
5. Click the **Branch** drop-down and select a branch or type the name of a new branch.
6. In the Path in Git Repo field, specify where in the repository to store your file.
Python notebooks have the suggested default file extension `.py`. If you use `.ipynb`, your notebook will save in iPython notebook format. If the file already exists on GitHub, you can directly copy and paste the URL of the file.
7. Click **Save** to finish linking your notebook. If this file did not previously exist, a prompt with the option **Save this file to your GitHub repo** displays.
8. Type a message and click **Save**.

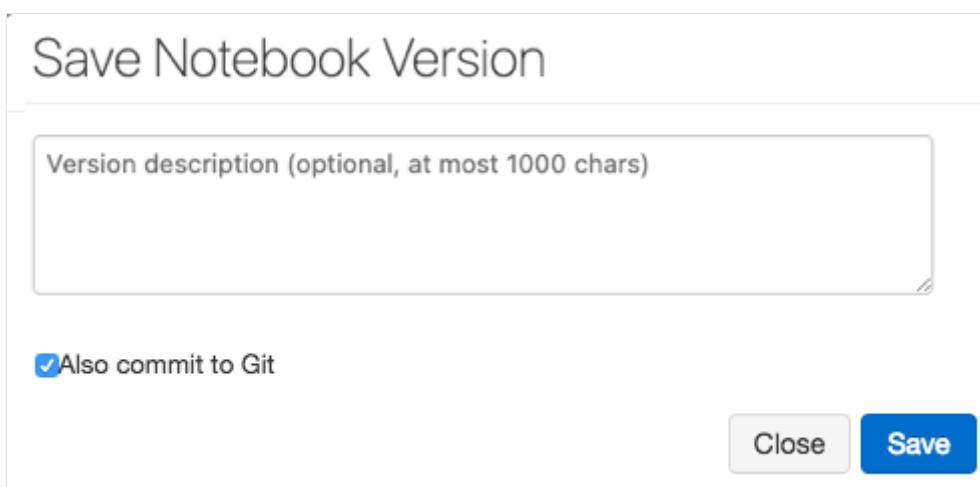
Save a notebook to GitHub

While the changes that you make to your notebook are saved automatically to the Azure Databricks version history, changes do not automatically persist to GitHub.

1. Click  in the right sidebar to open the history panel.



2. Click **Save Now** to save your notebook to GitHub. The Save Notebook Version dialog appears.
3. Optionally, enter a message to describe your change.
4. Make sure that **Also commit to Git** is selected.

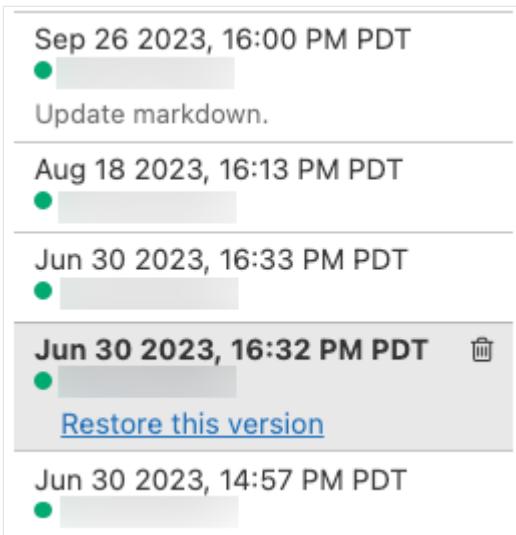


5. Click **Save**.

Revert or update a notebook to a version from GitHub

Once you link a notebook, Azure Databricks syncs your history with Git every time you re-open the history panel. Versions that sync to Git have commit hashes as part of the entry.

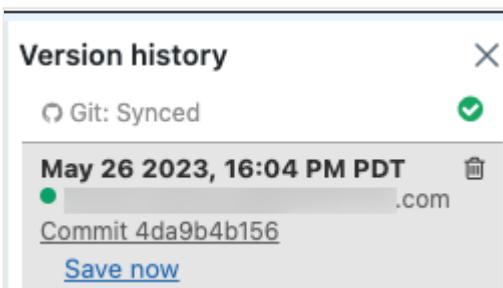
1. Click  in the right sidebar to open the history panel.



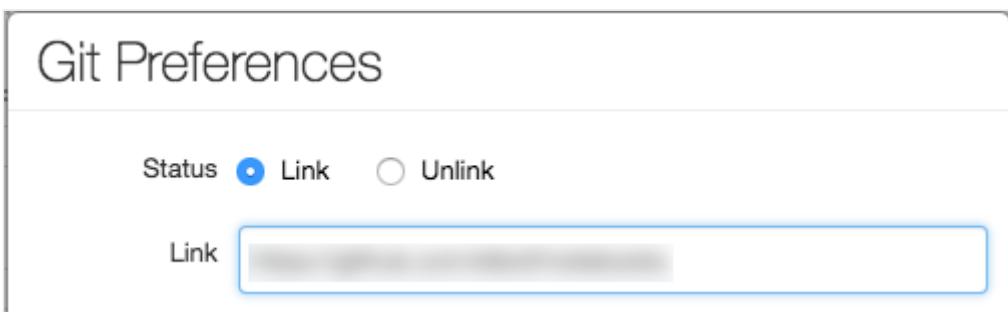
2. Choose an entry in the history panel. Azure Databricks displays that version.
3. Click **Restore this version**.
4. Click **Confirm** to confirm that you want to restore that version.

Unlink a notebook

1. Click  in the right sidebar to open the history panel.
2. The Git status bar displays **Git: Synced**.



3. Click **Git: Synced**.



4. In the Git Preferences dialog, click **Unlink**.
5. Click **Save**.

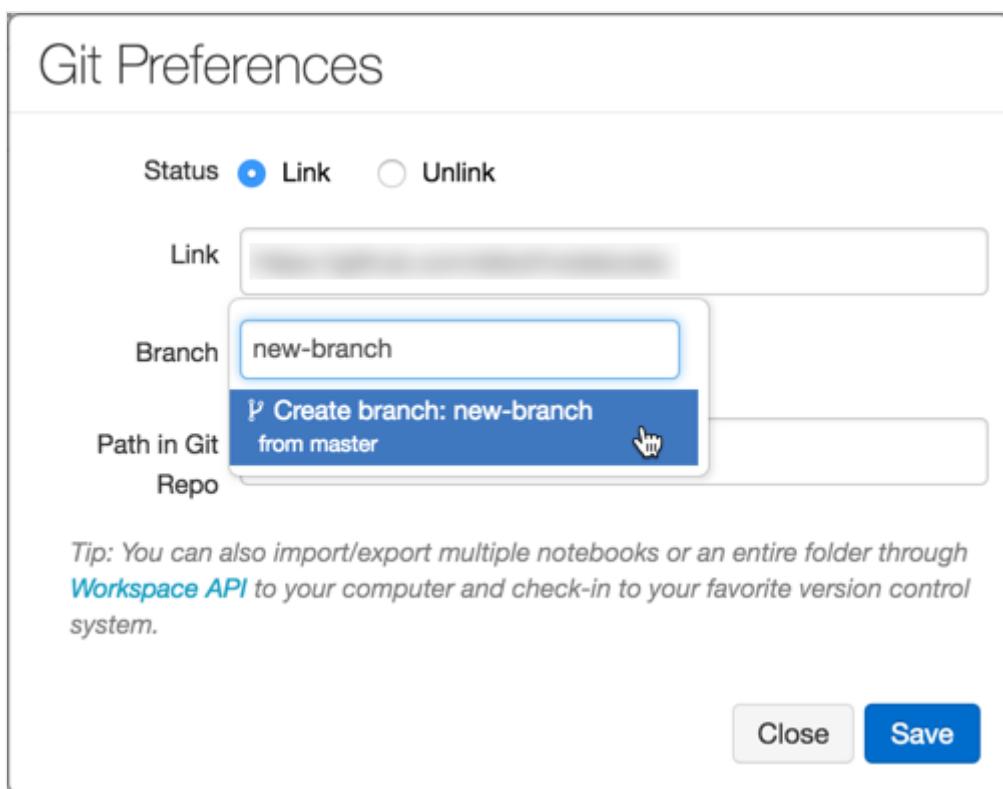
6. Click **Confirm** to confirm that you want to unlink the notebook from version control.

Use branches

You can work on any branch of your repository and create new branches inside Azure Databricks.

Create a branch

1. Click  in the right sidebar to open the history panel.
2. Click the Git status bar to open the GitHub panel.
3. Click the **Branch** dropdown.
4. Enter a branch name.

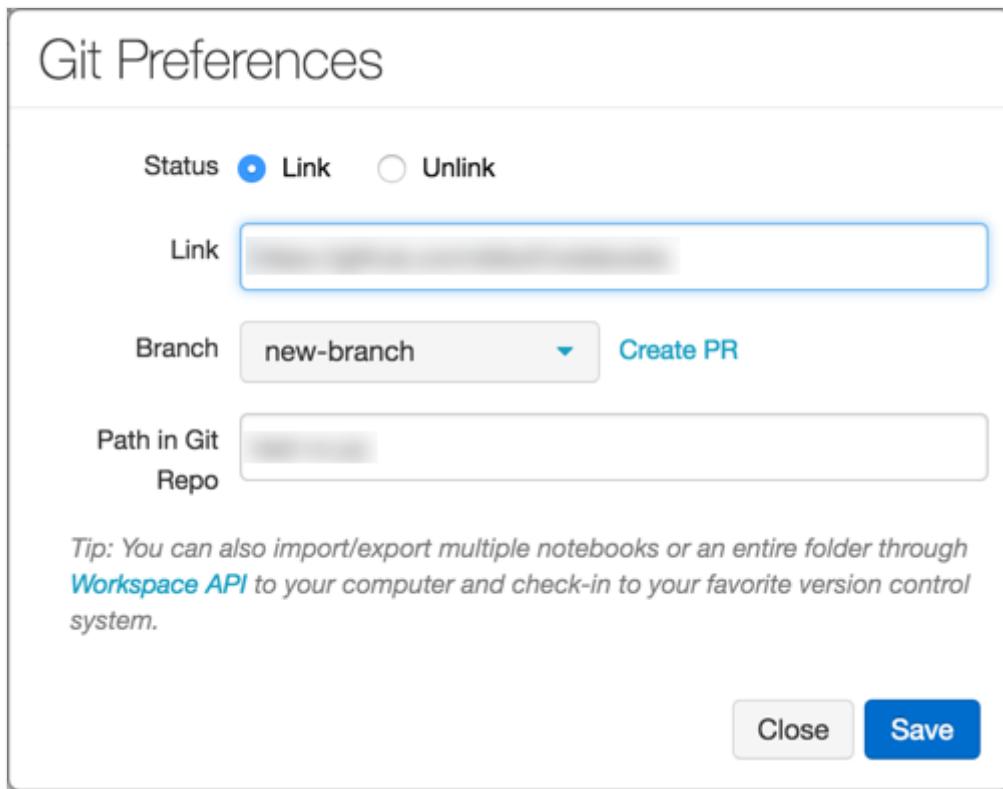


5. Select the **Create Branch** option at the bottom of the dropdown. The parent branch is indicated. You always branch from your current selected branch.

Create a pull request

1. Click  in the right sidebar to open the history panel.

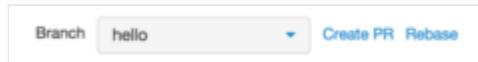
2. Click the Git status bar to open the GitHub panel.



3. Click Create PR. GitHub opens to a pull request page for the branch.

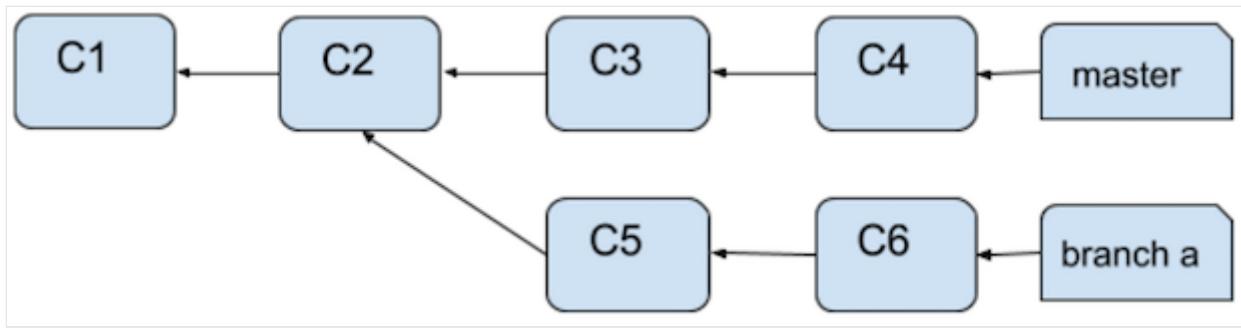
Rebase a branch

You can also rebase your branch inside Azure Databricks. The **Rebase** link displays if new commits are available in the parent branch. Only rebasing on top of the default branch of the parent repository is supported.

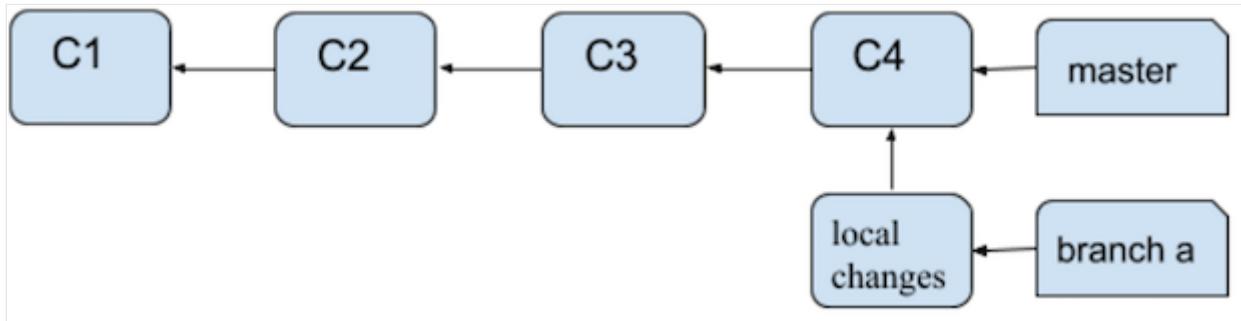


For example, assume that you are working on `databricks/reference-apps`. You fork it into your own account (for example, `brkyvz`) and start working on a branch called `my-branch`. If a new update is pushed to `databricks:master`, then the `Rebase` button displays, and you will be able to pull the changes into your branch `brkyvz:my-branch`.

Rebasing works a little differently in Azure Databricks. Assume the following branch structure:



After a rebase, the branch structure looks like:



What's different here is that Commits C5 and C6 do not apply on top of C4. They appear as local changes in your notebook. Merge conflicts show up as follows:

```

> 3 + 3

> <<<< 4f0ebd02fb668f7ecea5949a543c298620939ce6
"Hello World!"
=====
"Hello Universe!"
>>>> HEAD
  
```

You can then commit to GitHub once again using the **Save Now** button.

What happens if someone branched off from my branch that I just rebased?

If your branch (for example, `branch-a`) was the base for another branch (`branch-b`), and you rebase, you need not worry! Once a user also rebases `branch-b`, everything will work out. The best practice in this situation is to use separate branches for separate notebooks.

Best practices for code reviews

Azure Databricks supports Git branching.

- You can link a notebook to any branch in a repository. Azure Databricks recommends using a separate branch for each notebook.
- During development, you can link a notebook to a fork of a repository or to a non-default branch in the main repository. To integrate your changes upstream, you can use the **Create PR** link in the **Git Preferences** dialog in Azure Databricks to create a GitHub pull request. The Create PR link displays only if you're not working on the default branch of the parent repository.

Troubleshooting

If you receive errors related to syncing GitHub history, verify the following:

- You can only link a notebook to an initialized Git repository that isn't empty. Test the URL in a web browser.
- The GitHub personal access token must be active.
- To use a private GitHub repository, you must have permission to read the repository.
- If a notebook is linked to a GitHub branch that is renamed, the change is not automatically reflected in Azure Databricks. You must re-link the notebook to the branch manually.

Migrate to Databricks Repos

Users that need to migrate to Databricks Repos from the legacy Git version control can use the following guide:

- [Switching to Databricks Repos from Legacy Git integration ↗](#)

Continuous integration and delivery on Azure Databricks using Azure DevOps

Article • 01/12/2024

ⓘ Note

This article covers Azure DevOps, which is neither provided nor supported by Databricks. To contact the provider, see [Azure DevOps Services support](#).

Continuous integration and continuous delivery (CI/CD) refers to the process of developing and delivering software in short, frequent cycles through the use of automation pipelines.

Continuous integration begins with committing your code frequently to a branch in a source code repository. Each commit is merged with other developers' commits to ensure no conflicts are introduced. Changes are further validated by creating a build and running automated tests against that build. This process ultimately results in artifacts that are eventually deployed to a target, in this article's case an Azure Databricks workspace.

CI/CD development workflow

Databricks suggests the following workflow for CI/CD development with Azure DevOps:

1. Create a repository, or use an existing repository, with your third-party Git provider.
2. Connect your local development machine to the same third-party repository. For instructions, see your third-party Git provider's documentation.
3. Pull any existing updated artifacts (such as notebooks, code files, and build scripts) down to your local development machine from the third-party repository.
4. As necessary, create, update, and test artifacts on your local development machine. Then, push any new and changed artifacts from your local development machine to the third-party repository. For instructions, see your third-party Git provider's documentation.
5. Repeat steps 3 and 4 as needed.
6. Use Azure DevOps periodically as an integrated approach to automatically pulling artifacts from your third-party repository, building, testing, and running code on your Azure Databricks workspace, and reporting test and run results. While you can run Azure DevOps manually, in real-world implementations, you would instruct

your third-party Git provider to run Azure DevOps every time a specific event happens, such as a repository pull request.

There are numerous CI/CD tools you can use to manage and execute your pipeline. This article illustrates how to use [Azure DevOps](#). CI/CD is a design pattern, so the steps and stages outlined in this article's example should transfer with a few changes to the pipeline definition language in each tool. Furthermore, much of the code in this example pipeline is standard Python code that can be invoked in other tools.

Tip

For information about using Jenkins with Azure Databricks instead of Azure DevOps, see [CI/CD with Jenkins on Azure Databricks](#).

The rest of this article describes a pair of example pipelines in Azure DevOps that you can adapt to your own needs for Azure Databricks.

About the example

This article's example uses two pipelines to gather, deploy, and run some example Python code and Python notebooks that are stored in a remote Git repository.

The first pipeline, known as the *build* pipeline, prepares build artifacts for the second pipeline, known as the *release* pipeline. Separating the build pipeline from the release pipeline allows you to create a build artifact without deploying it or to simultaneously deploy artifacts from multiple builds.

In this example, you create the build and release pipelines, which do the following:

1. Creates an Azure virtual machine for the build pipeline.
2. Copies the files from your Git repository to the virtual machine.
3. Creates a gzip'ed tar file that contains the Python code, Python notebooks, and related build, deployment, and run settings files.
4. Copies the gzip'ed tar file as a zip file into a location for the release pipeline to access.
5. Creates another Azure virtual machine for the release pipeline.
6. Gets the zip file from the build pipeline's location and then unpackages the zip file to get the Python code, Python notebooks, and related build, deployment, and run settings files.
7. Deploys the Python code, Python notebooks, and related build, deployment, and run settings files to your remote Azure Databricks workspace.

8. Builds the Python wheel library's component code files into a Python wheel.
9. Runs unit tests on the component code to check the logic in the Python wheel.
10. Runs the Python notebooks, one of which calls the Python wheel's functionality.
11. Assesses the outcome of running the Python notebook, and publishes a related run results report.

Before you begin

To use this article's example, you must have:

- An existing [Azure DevOps](#) project. If you do not yet have a project, [create a project in Azure DevOps](#).
- An existing repository with a Git provider that Azure DevOps supports. You will add the Python example code, the example Python notebook, and related release settings files to this repository. If you do not yet have a repository, create one by following your Git provider's instructions. Then, connect your Azure DevOps project to this repository if you have not done so already. For instructions, follow the links in [Supported source repositories](#).

Step 1: Add the example's files to your repository

In this step, in the repository with your third-party Git provider, you add all of this article's example files that your Azure DevOps pipelines build, deploy, and run on your remote Azure Databricks workspace.

Step 1.1: Add the Python wheel component files

In this article's example, your Azure DevOps pipelines build and unit test a Python wheel. An Azure Databricks notebook then calls the built Python wheel's functionality.

To define the logic and unit tests for the Python wheel that the notebooks run against, in the root of your repository create two files named `addcol.py` and `test_addcol.py`, and add them to a folder structure named `python/dabdemo/dabdemo` in a `Libraries` folder, visualized as follows:

```
```\n    ``-- Libraries\n        ``-- python\n            ``-- dabdemo
```

```
```
`-- dabdemo
  |-- addcol.py
  `-- test_addcol.py
```

The `addcol.py` file contains a library function that is built later into a Python wheel and then installed on Azure Databricks clusters. It is a simple function that adds a new column, populated by a literal, to an Apache Spark DataFrame:

Python

```
# Filename: addcol.py
import pyspark.sql.functions as F

def with_status(df):
    return df.withColumn("status", F.lit("checked"))
```

The `test_addcol.py` file contains tests to pass a mock DataFrame object to the `with_status` function, defined in `addcol.py`. The result is then compared to a DataFrame object containing the expected values. If the values match, the test passes:

Python

```
# Filename: test_addcol.py
import pytest
from pyspark.sql import SparkSession
from dabdemo.addcol import *

class TestAppendCol(object):

    def test_with_status(self):
        spark = SparkSession.builder.getOrCreate()

        source_data = [
            ("paula", "white", "paula.white@example.com"),
            ("john", "baer", "john.baer@example.com")
        ]

        source_df = spark.createDataFrame(
            source_data,
            ["first_name", "last_name", "email"]
        )

        actual_df = with_status(source_df)

        expected_data = [
            ("paula", "white", "paula.white@example.com", "checked"),
            ("john", "baer", "john.baer@example.com", "checked")
        ]
        expected_df = spark.createDataFrame(
            expected_data,
```

```
        ["first_name", "last_name", "email", "status"]  
    )  
  
    assert(expected_df.collect() == actual_df.collect())
```

To enable the Databricks CLI to correctly package this library code into a Python wheel, create two files named `__init__.py` and `__main__.py` in the same folder as the preceding two files. Also, create a file named `setup.py` in the `python/dabdemo` folder, visualized as follows:

```
```  
`-- Libraries
 `-- python
 `-- dabdemo
 |-- dabdemo
 | |-- __init__.py
 | |-- __main__.py
 | |-- addcol.py
 | `-- test_addcol.py
 `-- setup.py
```

The `__init__.py` file contains the library's version number and author. Replace `<my-author-name>` with your name:

```
Python

Filename: __init__.py
__version__ = '0.0.1'
__author__ = '<my-author-name>'

import sys, os

sys.path.append(os.path.join(os.path.dirname(__file__), "..", ".."))
```

The `__main__.py` file contains the library's entry point:

```
Python

Filename: __main__.py
import sys, os

sys.path.append(os.path.join(os.path.dirname(__file__), "..", ".."))

from addcol import *

def main():
 pass
```

```
if __name__ == "__main__":
 main()
```

The `setup.py` file contains additional settings for building the library into a Python wheel. Replace `<my-url>`, `<my-author-name>@<my-organization>`, and `<my-package-description>` with valid values:

Python

```
Filename: setup.py
from setuptools import setup, find_packages

import dabdemo

setup(
 name = "dabdemo",
 version = dabdemo.__version__,
 author = dabdemo.__author__,
 url = "https://<my-url>",
 author_email = "<my-author-name>@<my-organization>",
 description = "<my-package-description>",
 packages = find_packages(include = ["dabdemo"]),
 entry_points={"group_1": "run=dabdemo.__main__:main"},
 install_requires = ["setuptools"]
)
```

## Step 1.2: Add a unit testing notebook for the Python wheel

Later on, the Databricks CLI runs a notebook job. This job runs a Python notebook with the filename of `run-unit-test.py`. This notebook runs `pytest` against the Python wheel library's logic.

To run the unit tests for this article's example, add to the root of your repository a notebook file named `run_unit_tests.py` with the following contents:

Python

```
Databricks notebook source

COMMAND ----

MAGIC %sh
MAGIC
MAGIC mkdir -p
"/Workspace${WORKSPACEBUNDLEPATH}/Validation/reports/junit/test-reports"
```

```
COMMAND ----

Prepare to run pytest.
import sys, pytest, os

Skip writing pyc files on a readonly filesystem.
sys.dont_write_bytecode = True

Run pytest.
retcode = pytest.main(["--junit-xml",
f"/Workspace{os.getenv('WORKSPACEBUNDLEPATH')}/Validation/reports/junit/test-reports/TEST-libout.xml",
f"/Workspace{os.getenv('WORKSPACEBUNDLEPATH')}/files/Libraries/python/dabdemo/dabdemo/"])

Fail the cell execution if there are any test failures.
assert retcode == 0, "The pytest invocation failed. See the log for details."
```

## Step 1.3: Add a notebook that calls the Python wheel

Later on, the Databricks CLI runs another notebook job. This notebook creates a DataFrame object, passes it to the Python wheel library's `with_status` function, prints the result, and report the job's run results. Create the root of your repository a notebook file named `dabdaddemo_notebook.py` with the following contents:

Python

```
Databricks notebook source

COMMAND ----

Restart Python after installing the Python wheel.
dbutils.library.restartPython()

COMMAND ----

from dabdemo.addcol import with_status

df = (spark.createDataFrame(
 schema = ["first_name", "last_name", "email"],
 data = [
 ("paula", "white", "paula.white@example.com"),
 ("john", "baer", "john.baer@example.com")
]
))

new_df = with_status(df)
```

```
display(new_df)

Expected output:
#
+-----+-----+-----+-----+
| first_name | last_name | email | status |
+-----+-----+-----+-----+
| paula | white | paula.white@example.com | checked |
+-----+-----+-----+-----+
| john | baer | john.baer@example.com | checked |
+-----+-----+-----+-----+
```

## Step 1.4: Add Python code that evaluates the notebook's run results

In a later step, the Databricks CLI will run a Python file job. This Python file's name is `evaluate_notebook_runs.py`. This file will determine the failure and success criteria for the notebook job run and report this failure or success result. Create in the root of your repository a file named `evaluate_notebook_runs.py` with the following contents:

Python

```
import unittest
import xmlrunner
import json
import glob
import os

class TestJobOutput(unittest.TestCase):

 test_output_path =
 f"/Workspace${os.getenv('WORKSPACEBUNDLEPATH')}/Validation/Output"

 def test_performance(self):
 path = self.test_output_path
 statuses = []

 for filename in glob.glob(os.path.join(path, '*.json')):
 print('Evaluating: ' + filename)

 with open(filename) as f:
 data = json.load(f)

 duration = data['tasks'][0]['execution_duration']

 if duration > 100000:
 status = 'FAILED'
 else:
 status = 'SUCCESS'
```

```

 statuses.append(status)
 f.close()

 self.assertFalse('FAILED' in statuses)

def test_job_run(self):
 path = self.test_output_path
 statuses = []

 for filename in glob.glob(os.path.join(path, '*.json')):
 print('Evaluating: ' + filename)

 with open(filename) as f:
 data = json.load(f)
 status = data['state']['result_state']
 statuses.append(status)
 f.close()

 self.assertFalse('FAILED' in statuses)

if __name__ == '__main__':
 unittest.main(
 testRunner = xmlrunner.XMLTestRunner(
 output =
f"/Workspace${os.getenv('WORKSPACEBUNDLEPATH')}/Validation/Output/test-
results",
),
 failfast = False,
 buffer = False,
 catchbreak = False,
 exit = False
)

```

## Step 1.5: Create the bundle configuration

This article's example uses *Databricks Asset Bundles* to define the settings and behaviors for building, deploying, and running the Python wheel, the two notebooks, and the Python code file. Databricks Asset Bundles, known simply as *bundles*, make it possible to express complete data, analytics, and ML projects as a collection of source files. See [What are Databricks Asset Bundles?](#).

To configure the bundle for this article's example, create in the root of your repository a file named `databricks.yml`. In this example `databricks.yml` file, replace the following placeholders:

- Replace `<bundle-name>` with a unique programmatic name for the bundle. For example, `azure-devops-demo`.

- Replace `<job-prefix-name>` with some string to help uniquely identify the jobs that are created in your Azure Databricks workspace for this example. For example, `azure-devops-demo`.
- Replace `<spark-version-id>` with the Databricks Runtime version ID for your job clusters, for example `13.3.x-scala2.12`.
- Replace `<cluster-node-type-id>` with the cluster node type ID for your job clusters, for example `Standard_DS3_v2`.
- Notice that `dev` in the `targets` mapping specifies the host and the related deployment behaviors. In real-world implementations, you can give this target a different name in your own bundles.

Here are the contents of this example's `databricks.yml` file:

YAML

```
Filename: databricks.yml
bundle:
 name: <bundle-name>

variables:
 job_prefix:
 description: A unifying prefix for this bundle's job and task names.
 default: <job-prefix-name>
 spark_version:
 description: The cluster's Spark version ID.
 default: <spark-version-id>
 node_type_id:
 description: The cluster's node type ID.
 default: <cluster-node-type-id>

artifacts:
 dabdemo-wheel:
 type: whl
 path: ./Libraries/python/dabdemo

resources:
 jobs:
 run-unit-tests:
 name: ${var.job_prefix}-run-unit-tests
 tasks:
 - task_key: ${var.job_prefix}-run-unit-tests-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 notebook_task:
 notebook_path: ./run_unit_tests.py
 source: WORKSPACE
```

```

libraries:
 - pypi:
 package: pytest
run-dabdemo-notebook:
 name: ${var.job_prefix}-run-dabdemo-notebook
 tasks:
 - task_key: ${var.job_prefix}-run-dabdemo-notebook-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 notebook_task:
 notebook_path: ./dabdemo_notebook.py
 source: WORKSPACE
 libraries:
 - whl:
 "/Workspace${workspace.root_path}/files/Libraries/python/dabdemo/dist/dabdem
 o-0.0.1-py3-none-any.whl"
 evaluate-notebook-runs:
 name: ${var.job_prefix}-evaluate-notebook-runs
 tasks:
 - task_key: ${var.job_prefix}-evaluate-notebook-runs-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 spark_python_task:
 python_file: ./evaluate_notebook_runs.py
 source: WORKSPACE
 libraries:
 - pypi:
 package: unittest-xml-reporting

targets:
 dev:
 mode: development
 workspace:
 host: <databricks-host>

```

For more information about the `databricks.yml` file's syntax, see [Databricks Asset Bundle configurations](#).

## Step 2: Define the build pipeline

Azure DevOps provides a cloud-hosted user interface for defining the stages of your CI/CD pipeline using YAML. For more information about Azure DevOps and pipelines, see the [Azure DevOps documentation](#).

In this step, you use YAML markup to define the build pipeline, which builds a deployment artifact. To deploy the code to an Azure Databricks workspace, you specify this pipeline's build artifact as input into a release pipeline. You define this release pipeline later.

To run build pipelines, Azure DevOps provides cloud-hosted, on-demand execution agents that support deployments to Kubernetes, VMs, Azure Functions, Azure Web Apps, and many more targets. In this example, you use an on-demand agent to automate building the deployment artifact.

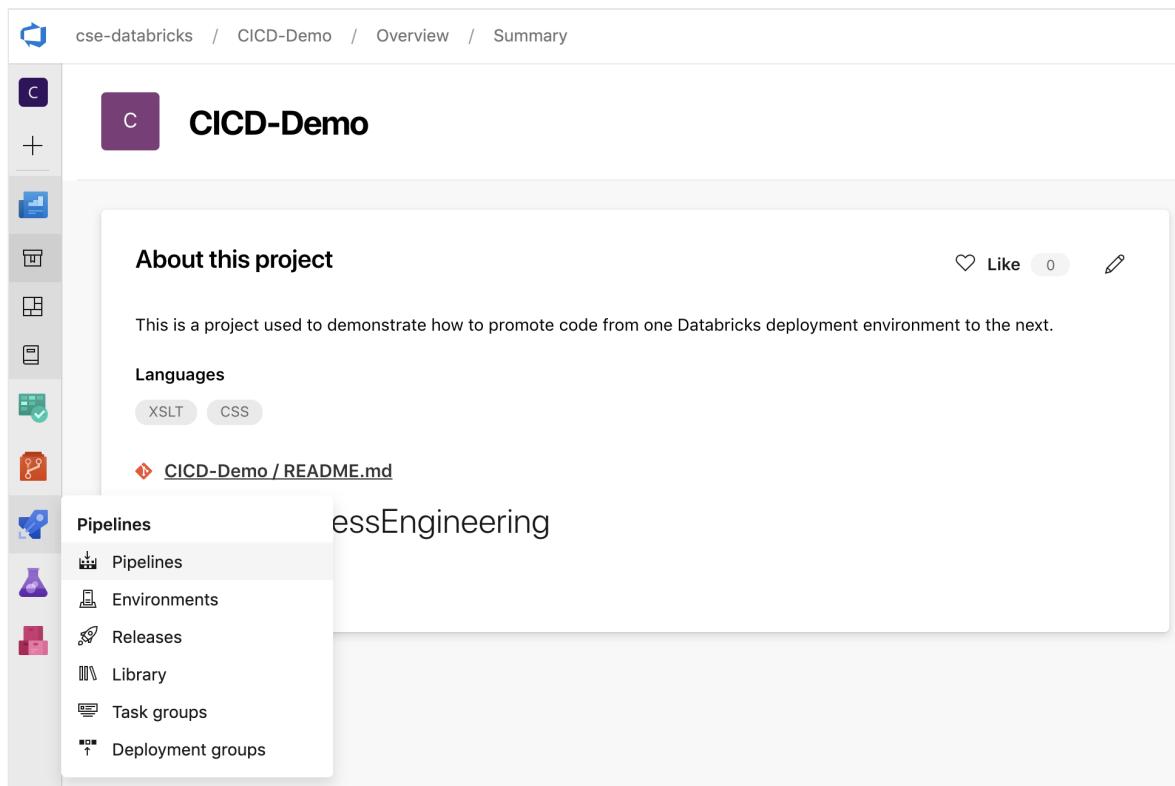
Define this article's example build pipeline as follows:

1. Sign in to [Azure DevOps](#) and then click the **Sign in** link to open your Azure DevOps project.

**! Note**

If the Azure Portal displays instead of your Azure DevOps project, click **More services > Azure DevOps organizations > My Azure DevOps organizations** and then open your Azure DevOps project.

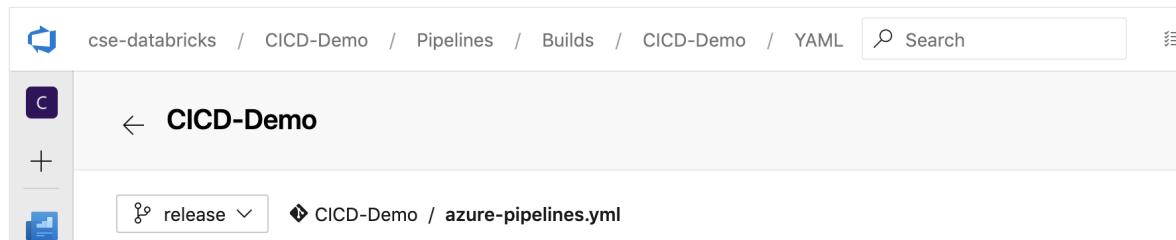
2. Click **Pipelines** in the sidebar, and then click **Pipelines** on the **Pipelines** menu.



3. Click the **New pipeline** button and follow the on-screen instructions. At the end of these instructions, the pipeline editor opens. Here you define your build pipeline

script in the `azure-pipelines.yml` file that appears. If the pipeline editor is not visible at the end of the instructions, select the build pipeline's name and then click **Edit**.

You can use the Git branch selector  to customize the build process for each branch in your Git repository. It is a CI/CD best practice to not do production work directly in your repository's `main` branch. This example assumes a branch named `release` exists in the repository to be used instead of `main`.



The `azure-pipelines.yml` build pipeline script is stored by default in the root of the remote Git repository that you associate with the pipeline.

4. Overwrite your pipeline's `azure-pipelines.yml` file's starter contents with the following definition, and then click **Save**.

```
YAML

Specify the trigger event to start the build pipeline.
In this case, new code merged into the release branch initiates a new
build.
trigger:
- release

Specify the operating system for the agent that runs on the Azure
virtual
machine for the build pipeline (known as the build agent). The
virtual
machine image in this example uses the Ubuntu 22.04 virtual machine
image in the Azure Pipeline agent pool. See
#
https://learn.microsoft.com/azure/devops/pipelines/agents/hosted#softwa
re
pool:
 vmImage: ubuntu-22.04

Download the files from the designated branch in the remote Git
repository
onto the build agent.
steps:
- checkout: self
 persistCredentials: true
 clean: true
```

```

Generate the deployment artifact. To do this, the build agent gathers
all the new or updated code to be given to the release pipeline,
including the sample Python code, the Python notebooks,
the Python wheel library component files, and the related Databricks
asset
bundle settings.
Use git diff to flag files that were added in the most recent Git
merge.
Then add the files to be used by the release pipeline.
The implementation in your pipeline will likely be different.
The objective here is to add all files intended for the current
release.
- script: |
 git diff --name-only --diff-filter=AMR HEAD^1 HEAD | xargs -I '{}' cp --parents -r '{}' $(Build.BinariesDirectory)
 mkdir -p $(Build.BinariesDirectory)/Libraries/python/dabdemo/dabdemo
 cp $(Build.Repository.LocalPath)/Libraries/python/dabdemo/dabdemo/*.* $(Build.BinariesDirectory)/Libraries/python/dabdemo/dabdemo
 cp $(Build.Repository.LocalPath)/Libraries/python/dabdemo/setup.py $(Build.BinariesDirectory)/Libraries/python/dabdemo
 cp $(Build.Repository.LocalPath)/*.* $(Build.BinariesDirectory)
 displayName: 'Get Changes'

Create the deployment artifact and then publish it to the
artifact repository.
- task: ArchiveFiles@2
 inputs:
 rootFolderOrFile: '$(Build.BinariesDirectory)'
 includeRootFolder: false
 archiveType: 'zip'
 archiveFile:
 '$(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip'
 replaceExistingArchive: true

- task: PublishBuildArtifacts@1
 inputs:
 ArtifactName: 'DatabricksBuild'

```

## Step 3: Define the release pipeline

The release pipeline deploys the build artifacts from the build pipeline to an Azure Databricks environment. Separating the release pipeline in this step from the build pipeline in the preceding steps allows you to create a build without deploying it or to deploy artifacts from multiple builds simultaneously.

1. In your Azure DevOps project, on the **Pipelines** menu in the sidebar, click **Releases**.

2. Click **New > New release pipeline**.

3. On the side of the screen is a list of featured templates for common deployment patterns. For this example release pipeline, click **Empty job**.

4. In the **Artifacts** box on the side of the screen, click **+ Add**. In the **Add an artifact** pane, for **Source (build pipeline)**, select the build pipeline that you created earlier. Then click **Add**.

5. You can configure how the pipeline is triggered by clicking  to display triggering options on the side of the screen. If you want a release to be initiated automatically based on build artifact availability or after a pull request workflow, enable the appropriate trigger. For now, in this example, in the last step of this article you manually trigger the build pipeline and then the release pipeline.

6. Click **Save > OK**.

## Step 3.1: Define environment variables for the release pipeline

This example's release pipeline relies on the following three environment variables, which you can add by clicking **Add** in the **Pipeline variables** section on the **Variables** tab, with a **Scope of Stage 1**:

- `BUNDLE_TARGET`, which should match the `target` name in your `databricks.yml` file. In this article's example, this is `dev`.
- `DATABRICKS_HOST`, which represents the [per-workspace URL](#) of your Azure Databricks workspace, beginning with `https://`, for example `https://adb-  
<workspace-id>.<random-number>.azuredatabricks.net`. Do not include the trailing `/` after `.net`.
- `DATABRICKS_TOKEN`, which represents your Azure Databricks [personal access token](#) or [Microsoft Entra ID \(formerly Azure Active Directory\) token](#). To create a personal access token, do the following:

 **Note**

As a security best practice, when you authenticate with automated tools, systems, scripts, and apps, Databricks recommends that you use personal access tokens belonging to **service principals** instead of workspace users. To create tokens for service principals, see [Manage tokens for a service principal](#).

1. In your Azure Databricks workspace, click your Azure Databricks username in the top bar, and then select **User Settings** from the drop down.
2. Click **Developer**.
3. Next to **Access tokens**, click **Manage**.
4. Click **Generate new token**.
5. (Optional) Enter a comment that helps you to identify this token in the future, and change the token's default lifetime of 90 days. To create a token with no lifetime (not recommended), leave the **Lifetime (days)** box empty (blank).
6. Click **Generate**.
7. Copy the displayed token to a secure location, and then click **Done**.

 **Note**

Be sure to save the copied token in a secure location. Do not share your copied token with others. If you lose the copied token, you cannot regenerate that exact same token. Instead, you must repeat this procedure to create a new token. If you lose the copied token, or you believe that the token has been compromised, Databricks strongly recommends that you immediately delete that token from your workspace by clicking the trash can (**Revoke**) icon next to the token on the **Access tokens** page.

If you are not able to create or use tokens in your workspace, this might be because your workspace administrator has disabled tokens or has not given you permission to create or use tokens. See your workspace administrator or the following:

- [Enable or disable personal access token authentication for the workspace](#)
- [Personal access token permissions](#)

All pipelines >  New release pipeline

Save  Create release

Pipeline Tasks Variables Retention Options History

**Pipeline variables**

Variable groups

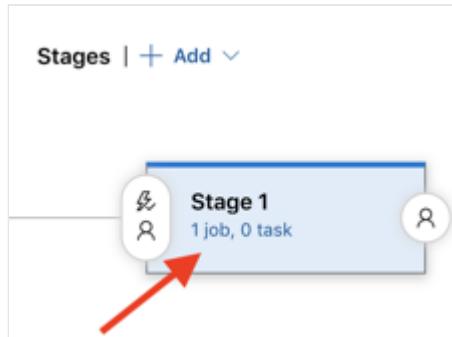
Predefined variables 

Name	Value	Scope
DATABRICKS_CLUSTER_ID	[REDACTED]	Stage 1
DATABRICKS_HOST	[REDACTED]	Stage 1
DATABRICKS_TOKEN	[REDACTED]	Stage 1

[+ Add](#)

## Step 3.2: Configure the release agent for the release pipeline

1. Click the **1 job, 0 task** link within the **Stage 1** object.



2. On the **Tasks** tab, click **Agent job**.
3. In the **Agent selection** section, for **Agent pool**, select **Azure Pipelines**.
4. For **Agent Specification**, select the same agent as you specified for the build agent earlier, in this example **ubuntu-22.04**.

Agent job ⓘ

Display name \*

Agent job

Agent selection ^

Agent pool ⓘ | [Pool information](#) | [Manage](#) ↗

Azure Pipelines

Agent Specification \*

ubuntu-22.04

5. Click **Save > OK**.

### Step 3.3: Set the Python version for the release agent

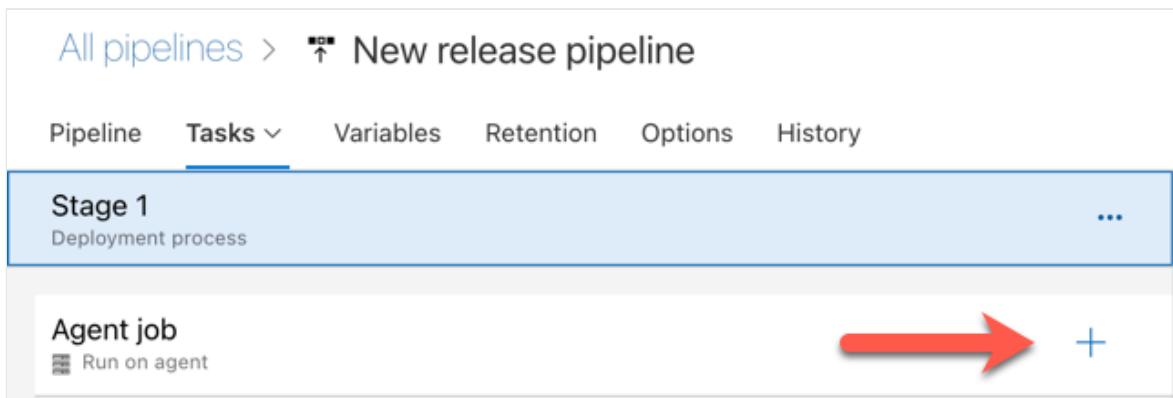
1. Click the plus sign in the **Agent job** section, indicated by the red arrow in the following figure. A searchable list of available tasks appears. There is also a **Marketplace** tab for third-party plug-ins that can be used to supplement the standard Azure DevOps tasks. You will add several tasks to the release agent during the next several steps.

All pipelines >  New release pipeline

Pipeline Tasks Variables Retention Options History

Stage 1 Deployment process ...

Agent job Run on agent 



2. The first task you add is **Use Python version**, located on the **Tool** tab. If you cannot find this task, use the **Search** box to look for it. When you find it, select it and then click the **Add** button next to the **Use Python version** task.

Add tasks | Refresh

Search

All Build Utility Test Package Deploy **Tool** Marketplace

- Java tool installer
- Kubectl tool installer
- Kubelogin tool installer
- Node.js tool installer
- NuGet tool installer
- Use .NET Core
- Use Python version**

Add

3. As with the build pipeline, you want to make sure that the Python version is compatible with the scripts called in subsequent tasks. In this case, click the **Use Python 3.x** task next to **Agent job**, and then set **Version spec** to **3.10**. Also set **Display name** to **Use Python 3.10**. This pipeline assumes that you are using Databricks Runtime 13.3 LTS on the clusters, which have Python 3.10.12 installed.

Use Python version ⓘ

Task version 0.\*

Display name \*

Use Python 3.10

Version spec \* ⓘ

3.10

4. Click **Save > OK**.

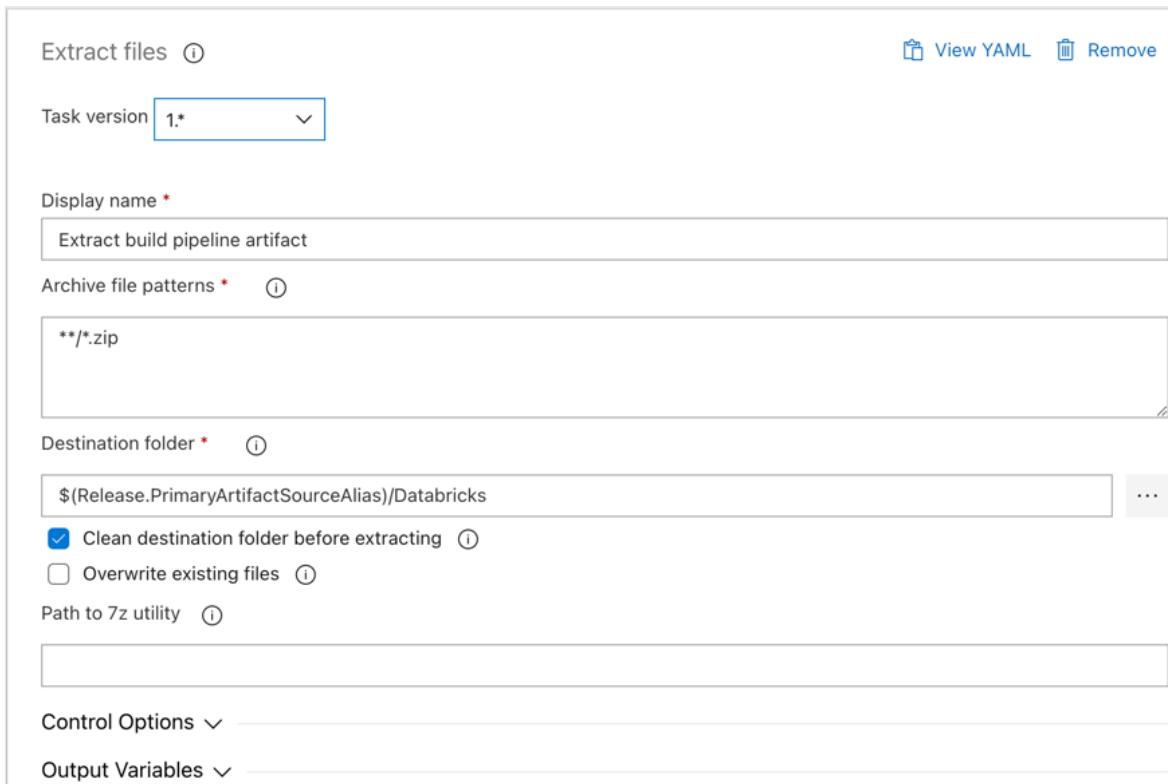
## Step 3.4: Unpackage the build artifact from the build pipeline

1. Next, have the release agent extract the Python wheel, related release settings files, the notebooks, and the Python code file from the zip file by using the **Extract files** task: click the plus sign in the **Agent job** section, select the **Extract files** task on the **Utility** tab, and then click **Add**.
2. Click the **Extract files** task next to **Agent job**, set **Archive file patterns** to `**/*.zip`, and set the **Destination folder** to the system variable `$(Release.PrimaryArtifactSourceAlias)/Databricks`. Also set **Display name** to `Extract build pipeline artifact`.

**① Note**

`$(Release.PrimaryArtifactSourceAlias)` represents an Azure DevOps-generated alias to identify the primary artifact source location on the release agent, for example `_<your-github-alias>.<your-github-repo-name>`. The release pipeline sets this value as the environment variable `RELEASE_PRIMARYARTIFACTSOURCEALIAS` in the **Initialize job** phase for the release agent. See [Classic release and artifacts variables](#).

3. Set **Display name** to `Extract build pipeline artifact`.



Extract files ①

Task version 1.\*

Display name \*

Extract build pipeline artifact

Archive file patterns \*

\*\*/\*.zip

Destination folder \*

`$(Release.PrimaryArtifactSourceAlias)/Databricks`

Clean destination folder before extracting ①

Overwrite existing files ①

Path to 7z utility ①

Control Options

Output Variables

4. Click **Save > OK**.

## Step 3.5: Set the `BUNDLE_ROOT` environment variable

For this article's example to operate as expected, you must set an environment variable named `BUNDLE_ROOT` in the release pipeline. Databricks Asset Bundles uses this environment variable to determine where the `databricks.yml` file is located. To set this environment variable:

1. Use the **Environment Variables** task: click the plus sign again in the **Agent job** section, select the **Environment Variables** task on the **Utility** tab, and then click **Add**.

 **Note**

If the **Environment Variables** task is not visible on the **Utility** tab, enter `Environment Variables` in the **Search** box and follow the on-screen instructions to add the task to the **Utility** tab. This might require you to leave Azure DevOps and then come back to this location where you left off.

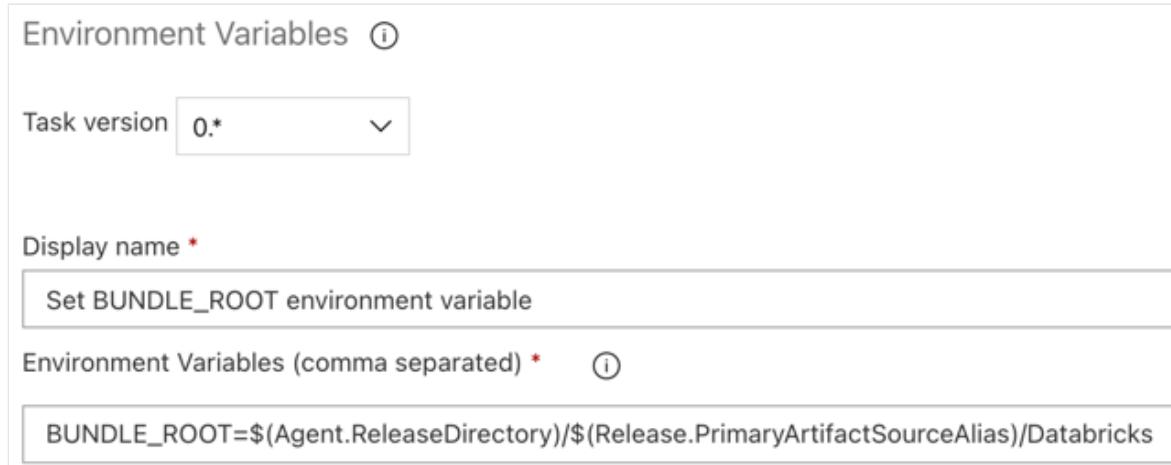
2. For **Environment Variables (comma separated)**, enter the following definition:

```
BUNDLE_ROOT=$(Agent.ReleaseDirectory)/$(Release.PrimaryArtifactSourceAlias)/Databricks
```

 **Note**

`$(Agent.ReleaseDirectory)` represents an Azure DevOps-generated alias to identify the release directory location on the release agent, for example `/home/vsts/work/r1/a`. The release pipeline sets this value as the environment variable `AGENT_RELEASEDIRECTORY` in the **Initialize job** phase for the release agent. See **Classic release and artifacts variables**. For information about `$(Release.PrimaryArtifactSourceAlias)`, see the note in the preceding step.

3. Set **Display name** to `Set BUNDLE_ROOT environment variable`.



The screenshot shows the configuration of the 'Environment Variables' task. The 'Task version' is set to '0.\*'. The 'Display name' is 'Set BUNDLE\_ROOT environment variable'. The 'Environment Variables (comma separated)' field contains the definition: `BUNDLE_ROOT=$(Agent.ReleaseDirectory)/$(Release.PrimaryArtifactSourceAlias)/Databricks`.

4. Click **Save > OK**.

## Step 3.6. Install the Databricks CLI, jq, Python wheel build tools, and unittest XML reporting

1. Next, install the Databricks CLI, the `jq` utility, Python wheel build tools, and the `unittest` XML reporting package on the release agent. The release agent will call the Databricks CLI, `jq`, Python wheel build tools, and `unittest` in the next few tasks. To do this, use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.
2. Click the **Bash Script** task next to **Agent job**.
3. For **Type**, select **Inline**.
4. Replace the contents of **Script** with the following command, which installs the Databricks CLI, the `jq` utility, Python wheel build tools, and the `unittest` XML reporting package:

```
Bash
```

```
curl -fsSL https://raw.githubusercontent.com/databricks/setup-
cli/main/install.sh | sh
sudo apt-get install jq
pip install wheel
pip install unittest-xml-reporting
```

5. Set **Display name** to `Install Databricks CLI, jq, Python wheel, and unittest XML reporting`.

Bash ⓘ

Task version 3.\*

Display name \*

Install Databricks CLI, jq, wheel, and unittest XML reporting

Type ⓘ

File Path  Inline

Script \*

```
curl -fsSL https://raw.githubusercontent.com/databricks/setup-cli/main/install.sh | sh
sudo apt-get install jq
pip install wheel
pip install unittest-xml-reporting
```

6. Click **Save > OK**.

## Step 3.7: Validate the Databricks Asset Bundle

In this step, you make sure that the `databricks.yml` file is syntactically correct.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.
2. Click the **Bash Script** task next to **Agent job**.
3. For **Type**, select **Inline**.
4. Replace the contents of **Script** with the following command, which uses the Databricks CLI to check whether the `databricks.yml` file is syntactically correct:

```
Bash
databricks bundle validate -t $(BUNDLE_TARGET)
```

5. Set **Display name** to `Validate bundle`.

6. Click **Save > OK**.

## Step 3.8: Deploy the bundle

In this step, you build the Python wheel and deploy the built Python wheel, the two Python notebooks, and the Python file from the release pipeline to your Azure Databricks workspace.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.
2. Click the **Bash Script** task next to **Agent job**.
3. For **Type**, select **Inline**.
4. Replace the contents of **Script** with the following command, which uses the Databricks CLI to build the Python wheel and to deploy this article's example files from the release pipeline to your Azure Databricks workspace:

```
Bash
```

```
databricks bundle deploy -t $(BUNDLE_TARGET)
```

5. Set **Display name** to **Deploy bundle**.

6. Click **Save > OK**.

## Step 3.9: Run the unit test notebook for the Python wheel

In this step, you run a job that runs the unit test notebook in your Azure Databricks workspace. This notebook runs unit tests against the Python wheel library's logic.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.
2. Click the **Bash Script** task next to **Agent job**.
3. For **Type**, select **Inline**.
4. Replace the contents of **Script** with the following command, which uses the Databricks CLI to run the job in your Azure Databricks workspace:

```
Bash
```

```
databricks bundle run -t $(BUNDLE_TARGET) run-unit-tests
```

5. Set **Display name** to `Run unit tests`.

6. Click **Save > OK**.

## Step 3.10: Run the notebook that calls the Python wheel

In this step, you run a job that runs another notebook in your Azure Databricks workspace. This notebook calls the Python wheel library.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.

2. Click the **Bash Script** task next to **Agent job**.

3. For **Type**, select **Inline**.

4. Replace the contents of **Script** with the following command, which uses the Databricks CLI to run the job in your Azure Databricks workspace:

```
Bash
```

```
databricks bundle run -t $(BUNDLE_TARGET) run-dabdemo-notebook
```

5. Set **Display name** to `Run notebook`.

6. Click **Save > OK**.

## Step 3.11: Evaluate the notebook run's test results

In this step, you run a job that runs a Python file in your Azure Databricks workspace. This Python file reports the results of the notebook run.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.

2. Click the **Bash Script** task next to **Agent job**.

3. For **Type**, select **Inline**.

4. Replace the contents of **Script** with the following command, which uses the Databricks CLI to run the job in your Azure Databricks workspace:

```
Bash
```

```
databricks bundle run -t $(BUNDLE_TARGET) evaluate-notebook-runs
```

5. Set **Display name** to `Evaluate notebook runs`.

6. Click **Save > OK**.

## Step 3.12: Import the test results

In this step, you copy from the previous step the reports that are generated, from your Azure Databricks workspace over into the release pipeline.

1. Use the **Bash** task: click the plus sign again in the **Agent job** section, select the **Bash** task on the **Utility** tab, and then click **Add**.
2. Click the **Bash Script** task next to **Agent job**.
3. For **Type**, select **Inline**.
4. Replace the contents of **Script** with the following commands. These commands use the Databricks CLI to get the path to the reports in your Azure Databricks workspace and then copy the reports from that path over into the path in the release pipeline where this article's example files are stored:

```
Bash
```

```
DATABRICKS_BUNDLE_WORKSPACE_FILE_PATH=$(databricks bundle validate -t
$BUNDLE_TARGET | jq -r .workspace.file_path)
databricks workspace export-dir \
$DATABRICKS_BUNDLE_WORKSPACE_FILE_PATH/Validation/Output/test-results \
$BUNDLE_ROOT/Validation/Output/test-results \
-t $BUNDLE_TARGET
```

5. Set **Display name** to `Evaluate notebook runs`.

6. Click **Save > OK**.

## Step 3.13: Publish the test results

Use the **Publish Test Results** task to archive the JSON results and publish the test results to Azure DevOps Test Hub. This enables you to visualize reports and dashboards related to the status of the test runs.

1. Add a **Publish Test Results** task to the release pipeline: click the plus sign again in the **Agent job** section, select the **Publish Test Results** task on the **Test** tab, and

then click **Add**.

2. Click the **\*\*Publish Test Results /TEST-\*.xml** task next to **Agent job**.

3. Leave all of the default settings unchanged.

Publish Test Results ⓘ

Task version 2.\*

Display name \*

Publish Test Results \*\*/TEST-\*.xml

Test result format \*

JUnit

Test results files \*

\*\*/TEST-\*.xml

Search folder ⓘ

\$(System.DefaultWorkingDirectory)

Merge test results ⓘ

Fail if there are test failures ⓘ

Test run title ⓘ

Advanced

Control Options

Output Variables

### ⓘ Note

`$(System.DefaultWorkingDirectory)` represents the local path on the agent where your source code files are downloaded, for example `/home/vsts/work/r1/a`. The release pipeline sets this value as the environment variable `SYSTEM_DEFAULTWORKINGDIRECTORY` in the **Initialize job** phase for the release agent. See [Use predefined variables](#).

4. Click **Save > OK**.

You have now completed configuring your release pipeline. It should look as follows:

## Agent job

 Run on agent



Use Python 3.10

Use Python version



Extract build pipeline artifact

Extract files



Set BUNDLE\_ROOT environment variable

Environment Variables



Install Databricks CLI, jq, wheel, and unittest XML report...

Bash



Validate bundle

Bash



Deploy bundle

Bash



Run unit tests

Bash



Run notebook

Bash



Evaluate notebook runs

Bash



Import test results

Bash



Publish Test Results \*\*/TEST-\*.xml

Publish Test Results

## Step 4: Run the build and release pipelines

In this step, you run the pipelines manually. To learn how to run the pipelines automatically, see [Specify events that trigger pipelines](#) and [Release triggers](#).

To run the build pipeline manually:

1. On the **Pipelines** menu in the sidebar, click **Pipelines**.
2. Click your build pipeline's name, and then click **Run pipeline**.
3. For **Branch/tag**, select the name of the branch in your Git repository that contains all of the source code that you added. This example assumes that this is in the `release` branch.
4. Click **Run**. The build pipeline's run page appears.

5. To see the build pipeline's progress and to view the related logs, click the spinning icon next to **Job**.
6. After the **Job** icon turns to a green check mark, proceed to run the release pipeline.

To run the release pipeline manually:

1. After the build pipeline has run successfully, on the **Pipelines** menu in the sidebar, click **Releases**.
2. Click your release pipeline's name, and then click **Create release**.
3. To see the release pipeline's progress, in the list of releases, click the name of the latest release.
4. In the **Stages** box, click **Stage 1**, and click **Logs**.

To view the published test run results:

1. On the **Test Plans** menu in the sidebar, click **Runs**.
2. In the **Recent test runs** section, on the **Test runs** tab, double-click the latest log dashboard entries in the list.

# CI/CD with Jenkins on Azure Databricks

Article • 01/17/2024

## ⓘ Note

This article covers Jenkins, which is neither provided nor supported by Databricks. To contact the provider, see [Jenkins Help](#).

There are numerous CI/CD tools you can use to manage and run your CI/CD pipelines. This article illustrates how to use the [Jenkins](#) automation server. CI/CD is a design pattern, so the steps and stages outlined in this article should transfer with a few changes to the pipeline definition language in each tool. Furthermore, much of the code in this example pipeline runs standard Python code, which you can invoke in other tools. For an overview of CI/CD on Azure Databricks, see [What is CI/CD on Azure Databricks?](#).

For information about using Azure DevOps with Azure Databricks instead, see [Continuous integration and delivery on Azure Databricks using Azure DevOps](#).

## CI/CD development workflow

Databricks suggests the following workflow for CI/CD development with Jenkins:

1. Create a repository, or use an existing repository, with your third-party Git provider.
2. Connect your local development machine to the same third-party repository. For instructions, see your third-party Git provider's documentation.
3. Pull any existing updated artifacts (such as notebooks, code files, and build scripts) from the third-party repository down onto your local development machine.
4. As desired, create, update, and test artifacts on your local development machine. Then push any new and changed artifacts from your local development machine up into the third-party repository. For instructions, see your third-party Git provider's documentation.
5. Repeat steps 3 and 4 as needed.
6. Use Jenkins periodically as an integrated approach to automatically pulling artifacts from your third-party repository down onto your local development machine or Azure Databricks workspace; building, testing, and running code on your local development machine or Azure Databricks workspace; and reporting test and run results. While you can run Jenkins manually, in real-world

implementations you would instruct your third-party Git provider to run Jenkins every time a specific event happens, such as a repository pull request.

The rest of this article uses an example project to describe one way to use Jenkins to implement the preceding CI/CD development workflow.

For information about using Azure DevOps instead of Jenkins, see [Continuous integration and delivery on Azure Databricks using Azure DevOps](#).

## Local development machine setup

This article's example uses Jenkins to instruct the [Databricks CLI](#) and [Databricks Asset Bundles](#) to do the following:

1. Build a Python wheel on your local development machine.
2. Deploy the built Python wheel along with additional Python files and Python notebooks from your local development machine to an Azure Databricks workspace.
3. Test and run the uploaded Python wheel and notebooks in that workspace.

To set up your local development machine to instruct your Azure Databricks workspace to perform the build and upload stages for this example, do the following on your local development machine:

### Step 1: Install required tools

In this step, you install the Databricks CLI, Jenkins, `jq`, and Python wheel build tools on your local development machine. These tools are required to run this example.

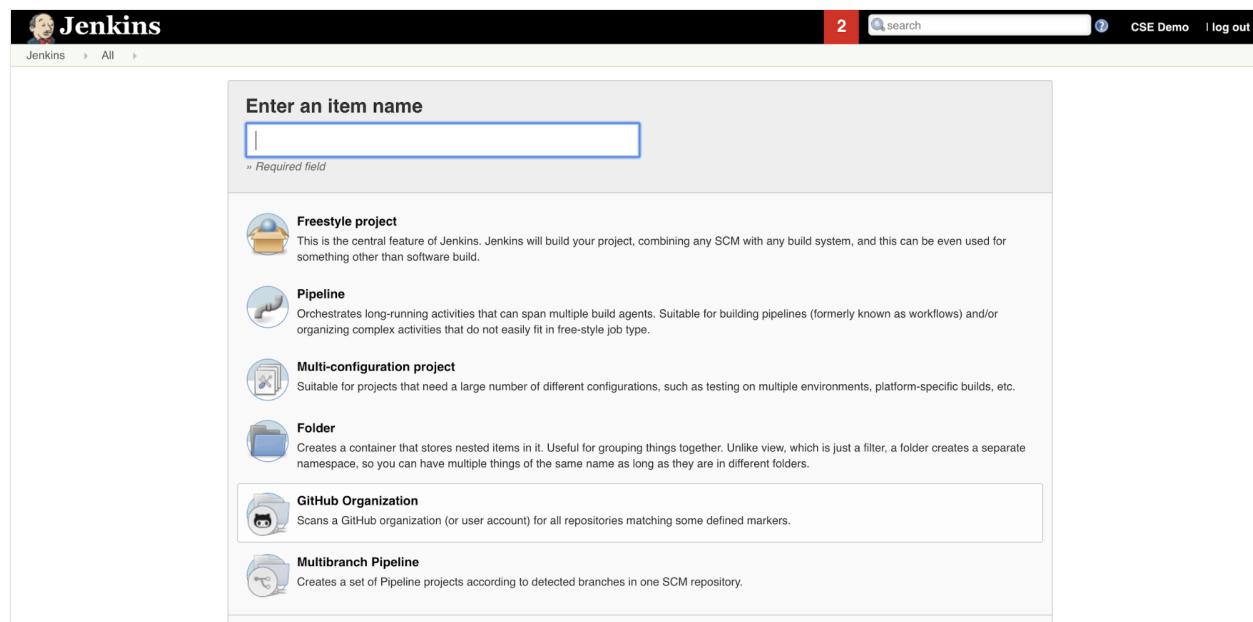
1. Install Databricks CLI version 0.205 or above, if you have not done so already. Jenkins uses the Databricks CLI to pass this example's test and run instructions on to your workspace. See [Install or update the Databricks CLI](#).
2. Install and start Jenkins, if you have not done so already. See [Installing Jenkins](#) for [Linux](#), [macOS](#), or [Windows](#).
3. [Install jq](#). This example uses `jq` to parse some JSON-formatted command output.
4. Use `pip` to install the [Python wheel build tools](#) with the following command (some systems might require you to use `pip3` instead of `pip`):

Bash

```
pip install --upgrade wheel
```

## Step 2: Create a Jenkins Pipeline

In this step, you use Jenkins to create a Jenkins Pipeline for this article's example. Jenkins provides a few different project types to create CI/CD pipelines. Jenkins Pipelines provide an interface to define stages in a Jenkins Pipeline by using [Groovy](#) code to call and configure Jenkins plugins.



To create the Jenkins Pipeline in Jenkins:

1. After you start Jenkins, from your Jenkins **Dashboard**, click **New Item**.
2. For **Enter an item name**, type a name for the Jenkins Pipeline, for example `jenkins-demo`.
3. Click the **Pipeline** project type icon.
4. Click **OK**. The Jenkins Pipeline's **Configure** page appears.
5. In the **Pipeline** area, in the **Definition** drop-down list, select **Pipeline script from SCM**.
6. In the **SCM** drop-down list, select **Git**.
7. For **Repository URL**, type the URL to the repository that is hosted by your third-part Git provider.
8. For **Branch Specifier**, type `*/<branch-name>`, where `<branch-name>` is the name of the branch in your repository that you want to use, for example `*/main`.
9. For **Script path**, type `Jenkinsfile`, if it is not already set. You create the `Jenkinsfile` later in this article.
10. Uncheck the box titled **Lightweight checkout**, if it is already checked.
11. Click **Save**.

## Step 3: Add global environment variables to Jenkins

In this step, you add three global environment variables to Jenkins. Jenkins passes these environment variables on to the Databricks CLI. The Databricks CLI needs the values for these environment variables to authenticate with your Azure Databricks workspace. This example uses OAuth machine-to-machine (M2M) authentication for a service principal (although other authentication types are also available). To set up OAuth M2M authentication for your Azure Databricks workspace, see [OAuth machine-to-machine \(M2M\) authentication](#).

The three global environment variables for this example are:

- `DATABRICKS_HOST`, set to your Azure Databricks workspace URL, starting with `https://`. See [Workspace instance names, URLs, and IDs](#).
- `DATABRICKS_CLIENT_ID`, set to the service principal's client ID, which is also known as its application ID.
- `DATABRICKS_CLIENT_SECRET`, set to the service principal's Azure Databricks OAuth secret.

To set global environment variables in Jenkins, from your Jenkins **Dashboard**:

1. In the sidebar, click **Manage Jenkins**.
2. In the **System Configuration** section, click **System**.
3. In the **Global properties** section, check the box titled **Environment variables**.
4. Click **Add** and then enter the environment variable's **Name** and **Value**. Repeat this for each additional environment variable.
5. When you are finished adding environment variables, click **Save** to return to your Jenkins **Dashboard**.

## Design the Jenkins Pipeline

Jenkins provides a few different project types to create CI/CD pipelines. This example implements a Jenkins Pipeline. Jenkins Pipelines provide an interface to define stages in a Jenkins Pipeline by using [Groovy](#) code to call and configure Jenkins plugins.

You write a Jenkins Pipeline definition in a text file called a *Jenkinsfile*, which in turn is checked into a project's source control repository. For more information, see [Jenkins Pipeline](#). Here is the Jenkins Pipeline for this article's example. In this example `Jenkinsfile`, replace the following placeholders:

- Replace `<user-name>` and `<repo-name>` with the username and repository name for your hosted by your third-part Git provider. This article uses a GitHub URL as an

example.

- Replace `<release-branch-name>` with the name of the release branch in your repository. For example, this could be `main`.
- Replace `<databricks-cli-installation-path>` with the path on your local development machine where the Databricks CLI is installed. For example, on macOS this could be `/usr/local/bin`.
- Replace `<jq-installation-path>` with the path on your local development machine where `jq` is installed. For example, on macOS this could be `/usr/local/bin`.
- Replace `<job-prefix-name>` with some string to help uniquely identify the Azure Databricks jobs that are created in your workspace for this example. For example, this could be `jenkins-demo`.
- Notice that `BUNDLETARGET` is set to `dev`, which is the name of the Databricks Asset Bundle target that is defined later in this article. In real-world implementations you would change this to the name of your own bundle target. More details about bundle targets are provided later in this article.

Here is the `Jenkinsfile`, which must be added to the root of your repository:

groovy

```
// Filename: Jenkinsfile
node {
 def GITREPOREMOTE = "https://github.com/<user-name>/<repo-name>.git"
 def GITBRANCH = "<release-branch-name>"
 def DBCLIPATH = "<databricks-cli-installation-path>"
 def JQPATH = "<jq-installation-path>"
 def JOBPREFIX = "<job-prefix-name>"
 def BUNDLETARGET = "dev"

 stage('Checkout') {
 git branch: GITBRANCH, url: GITREPOREMOTE
 }
 stage('Validate Bundle') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle validate -t ${BUNDLETARGET}
 """
 }
 stage('Deploy Bundle') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle deploy -t ${BUNDLETARGET}
 """
 }
 stage('Run Unit Tests') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} run-unit-
tests
 """
 }
}
```

```

stage('Run Notebook') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} run-dabdemo-
notebook
 """
}

stage('Evaluate Notebook Runs') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} evaluate-
notebook-runs
 """
}

stage('Import Test Results') {
 def DATABRICKS_BUNDLE_WORKSPACE_ROOT_PATH
 def getPath = "${DBCLIPATH}/databricks bundle validate -t
${BUNDLETARGET} | ${JQPATH}/jq -r .workspace.file_path"
 def output = sh(script: getPath, returnStdout: true).trim()

 if (output) {
 DATABRICKS_BUNDLE_WORKSPACE_ROOT_PATH = "${output}"
 } else {
 error "Failed to capture output or command execution failed:
${getPath}"
 }

 sh """#!/bin/bash
 ${DBCLIPATH}/databricks workspace export-dir \
 ${DATABRICKS_BUNDLE_WORKSPACE_ROOT_PATH}/Validation/Output/test-
results \
 ${WORKSPACE}/Validation/Output/test-results \
 -t ${BUNDLETARGET} \
 --overwrite
 """
}

stage('Publish Test Results') {
 junit allowEmptyResults: true, testResults: '**/test-results/*.xml',
skipPublishingChecks: true
}
}

```

The remainder of this article describes each stage in this Jenkins Pipeline and how to set up the artifacts and commands for Jenkins to run at that stage.

## Pull the latest artifacts from the third-party repository

The first stage in this Jenkins Pipeline, the `Checkout` stage, is defined as follows:

groovy

```
stage('Checkout') {
 git branch: GITBRANCH, url: GITREPOREMOTE
}
```

This stage makes sure that the working directory that Jenkins uses on your local development machine has the latest artifacts from your third-party Git repository. Typically, Jenkins sets this working directory to `<your-user-home-directory>/.jenkins/workspace/<pipeline-name>`. This enables you on the same local development machine to keep your own copy of artifacts in development separate from the artifacts that Jenkins uses from your third-party Git repository.

## Validate the Databricks Asset Bundle

The second stage in this Jenkins Pipeline, the `Validate Bundle` stage, is defined as follows:

```
groovy

stage('Validate Bundle') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle validate -t ${BUNDLETARGET}
 """
}
```

This stage makes sure that the Databricks Asset Bundle, which defines the workflows for testing and running your artifacts, is syntactically correct. *Databricks Asset Bundles*, known simply as *bundles*, make it possible to express complete data, analytics, and ML projects as a collection of source files. See [What are Databricks Asset Bundles?](#).

To define the bundle for this article, create a file named `databricks.yml` in the root of the cloned repository on your local machine. In this example `databricks.yml` file, replace the following placeholders:

- Replace `<bundle-name>` with a unique programmatic name for the bundle. For example, this could be `jenkins-demo`.
- Replace `<job-prefix-name>` with some string to help uniquely identify the Azure Databricks jobs that are created in your workspace for this example. For example, this could be `jenkins-demo`. It should match the `JOBPREFIX` value in your `Jenkinsfile`.
- Replace `<spark-version-id>` with the Databricks Runtime version ID for your job clusters, for example `13.3.x-scala2.12`.

- Replace `<cluster-node-type-id>` with the node type ID for your job clusters, for example `Standard_DS3_v2`.
- Notice that `dev` in the `targets` mapping is the same as the `BUNDLETARGET` in your Jenkinsfile. A bundle target specifies the host and the related deployment behaviors.

Here is the `databricks.yml` file, which must be added to the root of your repository for this example to operate correctly:

YAML

```
Filename: databricks.yml
bundle:
 name: <bundle-name>

variables:
 job_prefix:
 description: A unifying prefix for this bundle's job and task names.
 default: <job-prefix-name>
 spark_version:
 description: The cluster's Spark version ID.
 default: <spark-version-id>
 node_type_id:
 description: The cluster's node type ID.
 default: <cluster-node-type-id>

artifacts:
 dabdemo-wheel:
 type: whl
 path: ./Libraries/python/dabdemo

resources:
 jobs:
 run-unit-tests:
 name: ${var.job_prefix}-run-unit-tests
 tasks:
 - task_key: ${var.job_prefix}-run-unit-tests-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 notebook_task:
 notebook_path: ./run_unit_tests.py
 source: WORKSPACE
 libraries:
 - pypi:
 package: pytest
 run-dabdemo-notebook:
 name: ${var.job_prefix}-run-dabdemo-notebook
 tasks:
```

```

- task_key: ${var.job_prefix}-run-dabdemo-notebook-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 data_security_mode: SINGLE_USER
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 notebook_task:
 notebook_path: ./dabdemo_notebook.py
 source: WORKSPACE
 libraries:
 - whl:
 "/Workspace${workspace.root_path}/files/Libraries/python/dabdemo/dist/dabdem
o-0.0.1-py3-none-any.whl"
 evaluate-notebook-runs:
 name: ${var.job_prefix}-evaluate-notebook-runs
 tasks:
 - task_key: ${var.job_prefix}-evaluate-notebook-runs-task
 new_cluster:
 spark_version: ${var.spark_version}
 node_type_id: ${var.node_type_id}
 num_workers: 1
 spark_env_vars:
 WORKSPACEBUNDLEPATH: ${workspace.root_path}
 spark_python_task:
 python_file: ./evaluate_notebook_runs.py
 source: WORKSPACE
 libraries:
 - pypi:
 package: unittest-xml-reporting

targets:
 dev:
 mode: development

```

For more information about the `databricks.yml` file, see [Databricks Asset Bundle configurations](#).

## Deploy the bundle to your workspace

The Jenkins Pipeline's third stage, titled `Deploy Bundle`, is defined as follows:

```

groovy

stage('Deploy Bundle') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle deploy -t ${BUNDLETARGET}
 """
}

```

This stage does two things:

1. Because the `artifact` mapping in the `databricks.yml` file is set to `whl`, this instructs the Databricks CLI to build the Python wheel by using the `setup.py` file in the specified location.
2. After the Python wheel is built on your local development machine, the Databricks CLI deploys the built Python wheel along with the specified Python files and notebooks to your Azure Databricks workspace. By default, Databricks Asset Bundles deploys the Python wheel and other files to `/Workspace/Users/<your-username>/.bundle/<bundle-name>/<target-name>`.

To enable the Python wheel to be built as specified in the `databricks.yml` file, create the following folders and files in the root of your cloned repository on your local machine.

To define the logic and the unit tests for the Python wheel that the notebook will run against, create two files named `addcol.py` and `test_addcol.py`, and add them to a folder structure named `python/dabdemo/dabdemo` within your repository's `Libraries` folder, visualized as follows (ellipses indicate omitted folders in the repo, for brevity):

```
|-- ...
|-- Libraries
| '-- python
| '-- dabdemo
| '-- dabdemo
| |-- addcol.py
| '-- test_addcol.py
|-- ...
```

The `addcol.py` file contains a library function that is built later into a Python wheel and then installed on an Azure Databricks cluster. It is a simple function that adds a new column, populated by a literal, to an Apache Spark DataFrame:

Python

```
Filename: addcol.py
import pyspark.sql.functions as F

def with_status(df):
 return df.withColumn("status", F.lit("checked"))
```

The `test_addcol.py` file contains tests to pass a mock DataFrame object to the `with_status` function, defined in `addcol.py`. The result is then compared to a DataFrame

object containing the expected values. If the values match, which in this case they do, the test passes:

Python

```
Filename: test_addcol.py
import pytest
from pyspark.sql import SparkSession
from dabdemo.addcol import *

class TestAppendCol(object):

 def test_with_status(self):
 spark = SparkSession.builder.getOrCreate()

 source_data = [
 ("paula", "white", "paula.white@example.com"),
 ("john", "baer", "john.baer@example.com")
]

 source_df = spark.createDataFrame(
 source_data,
 ["first_name", "last_name", "email"]
)

 actual_df = with_status(source_df)

 expected_data = [
 ("paula", "white", "paula.white@example.com", "checked"),
 ("john", "baer", "john.baer@example.com", "checked")
]
 expected_df = spark.createDataFrame(
 expected_data,
 ["first_name", "last_name", "email", "status"]
)

 assert(expected_df.collect() == actual_df.collect())
```

To enable the Databricks CLI to correctly package this library code into a Python wheel, create two files named `__init__.py` and `__main__.py` in the same folder as the preceding two files. Also, create a file named `setup.py` in the `python/dabdemo` folder, visualized as follows (ellipses indicate omitted folders, for brevity):

```
-- ...
|-- Libraries
| |-- python
| |-- dabdemo
| |-- dabdemo
| |-- __init__.py
```

```
| | |-- __main__.py
| | |-- addcol.py
| | `-- test_addcol.py
| `-- setup.py
|-- ...
```

The `__init__.py` file contains the library's version number and author. Replace `<my-author-name>` with your name:

Python

```
Filename: __init__.py
__version__ = '0.0.1'
__author__ = '<my-author-name>'

import sys, os

sys.path.append(os.path.join(os.path.dirname(__file__), "..", ".."))
```

The `__main__.py` file contains the library's entry point:

Python

```
Filename: __main__.py
import sys, os

sys.path.append(os.path.join(os.path.dirname(__file__), "..", ".."))

from addcol import *

def main():
 pass

if __name__ == "__main__":
 main()
```

The `setup.py` file contains additional settings for building the library into a Python wheel. Replace `<my-url>`, `<my-author-name>@<my-organization>`, and `<my-package-description>` with meaningful values:

Python

```
Filename: setup.py
from setuptools import setup, find_packages

import dabdemo

setup(
```

```
 name = "dabdemo",
 version = dabdemo.__version__,
 author = dabdemo.__author__,
 url = "https://<my-url>",
 author_email = "<my-author-name>@<my-organization>",
 description = "<my-package-description>",
 packages = find_packages(include = ["dabdemo"]),
 entry_points={"group_1": "run=dabdemo.__main__:main"},
 install_requires = ["setuptools"]
)
```

## Test the Python wheel's component logic

The `Run Unit Tests` stage, the fourth stage of this Jenkins Pipeline, uses `pytest` to test a library's logic to make sure it works as built. This stage is defined as follows:

```
groovy

stage('Run Unit Tests') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} run-unit-tests
 """
}
```

This stage uses the Databricks CLI to run a notebook job. This job runs the Python notebook with the filename of `run-unit-test.py`. This notebook runs `pytest` against the library's logic.

To run the unit tests for this example, add a Python notebook file named `run_unit_tests.py` with the following contents to the root of your cloned repository on your local machine:

```
Python

Databricks notebook source

COMMAND ----

MAGIC %sh
MAGIC
MAGIC mkdir -p
"/Workspace${WORKSPACEBUNDLEPATH}/Validation/reports/junit/test-reports"

COMMAND ----

Prepare to run pytest.
import sys, pytest, os
```

```

Skip writing pyc files on a readonly filesystem.
sys.dont_write_bytecode = True

Run pytest.
retcode = pytest.main(["--junit-xml",
f"/Workspace{os.getenv('WORKSPACEBUNDLEPATH')}/Validation/reports/junit/test
-reports/TEST-libout.xml",
f"/Workspace{os.getenv('WORKSPACEBUNDLEPATH')}/files/Libraries/python/dabdem
o/dabdemo/"])

Fail the cell execution if there are any test failures.
assert retcode == 0, "The pytest invocation failed. See the log for
details."

```

## Use the built Python wheel

The fifth stage of this Jenkins Pipeline, titled `Run Notebook`, runs a Python notebook that calls the logic in the built Python wheel, as follows:

```

groovy

stage('Run Notebook') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} run-dabdemo-
notebook
 """
}

```

This stage runs the Databricks CLI, which in turn instructs your workspace to run a notebook job. This notebook creates a DataFrame object, passes it to the library's `with_status` function, prints the result, and report the job's run results. Create the notebook by adding a Python notebook file named `dabdaddemo_notebook.py` with the following contents in the root of your cloned repository on your local development machine:

```

Python

Databricks notebook source

COMMAND ----

Restart Python after installing the wheel.
dbutils.library.restartPython()

COMMAND ----

from dabdemo.addcol import with_status

```

```

df = (spark.createDataFrame(
 schema = ["first_name", "last_name", "email"],
 data = [
 ("paula", "white", "paula.white@example.com"),
 ("john", "baer", "john.baer@example.com")
]
))

new_df = with_status(df)

display(new_df)

Expected output:
#
+-----+-----+-----+-----+
| first_name | last_name | email | status |
+-----+-----+-----+-----+
| paula | white | paula.white@example.com | checked |
+-----+-----+-----+-----+
| john | baer | john.baer@example.com | checked |
+-----+-----+-----+

```

## Evaluate notebook job run results

The `Evaluate Notebook Runs` stage, the sixth stage of this Jenkins Pipeline, evaluates the results of the preceding notebook job run. This stage is defined as follows:

groovy

```

stage('Evaluate Notebook Runs') {
 sh """#!/bin/bash
 ${DBCLIPATH}/databricks bundle run -t ${BUNDLETARGET} evaluate-
notebook-runs
 """
}

```

This stage runs the Databricks CLI, which in turn instructs your workspace to run a Python file job. This Python file determines the failure and success criteria for the notebook job run and reports this failure or success result. Create a file named `evaluate_notebook_runs.py` with the following contents in the root of your cloned repository in your local development machine:

Python

```

import unittest
import xmlrunner
import json

```

```
import glob
import os

class TestJobOutput(unittest.TestCase):

 test_output_path =
f"/Workspace${os.getenv('WORKSPACEBUNDLEPATH')}/Validation/Output"

 def test_performance(self):
 path = self.test_output_path
 statuses = []

 for filename in glob.glob(os.path.join(path, '*.json')):
 print('Evaluating: ' + filename)

 with open(filename) as f:
 data = json.load(f)

 duration = data['tasks'][0]['execution_duration']

 if duration > 100000:
 status = 'FAILED'
 else:
 status = 'SUCCESS'

 statuses.append(status)
 f.close()

 self.assertFalse('FAILED' in statuses)

 def test_job_run(self):
 path = self.test_output_path
 statuses = []

 for filename in glob.glob(os.path.join(path, '*.json')):
 print('Evaluating: ' + filename)

 with open(filename) as f:
 data = json.load(f)
 status = data['state']['result_state']
 statuses.append(status)
 f.close()

 self.assertFalse('FAILED' in statuses)

if __name__ == '__main__':
 unittest.main(
 testRunner = xmlrunner.XMLTestRunner(
 output =
f"/Workspace${os.getenv('WORKSPACEBUNDLEPATH')}/Validation/Output/test-
results",
),
 failfast = False,
 buffer = False,
 catchbreak = False,
```

```
 exit = False
}
```

## Import and report test results

The seventh stage in this Jenkins Pipeline, titled `Import Test Results`, uses the Databricks CLI to send the test results from your workspace over to your local development machine. The eighth and final stage, titled `Publish Test Results`, publishes the test results to Jenkins by using the `junit` Jenkins plugin. This enables you to visualize reports and dashboards related to the status of the test results. These stages are defined as follows:

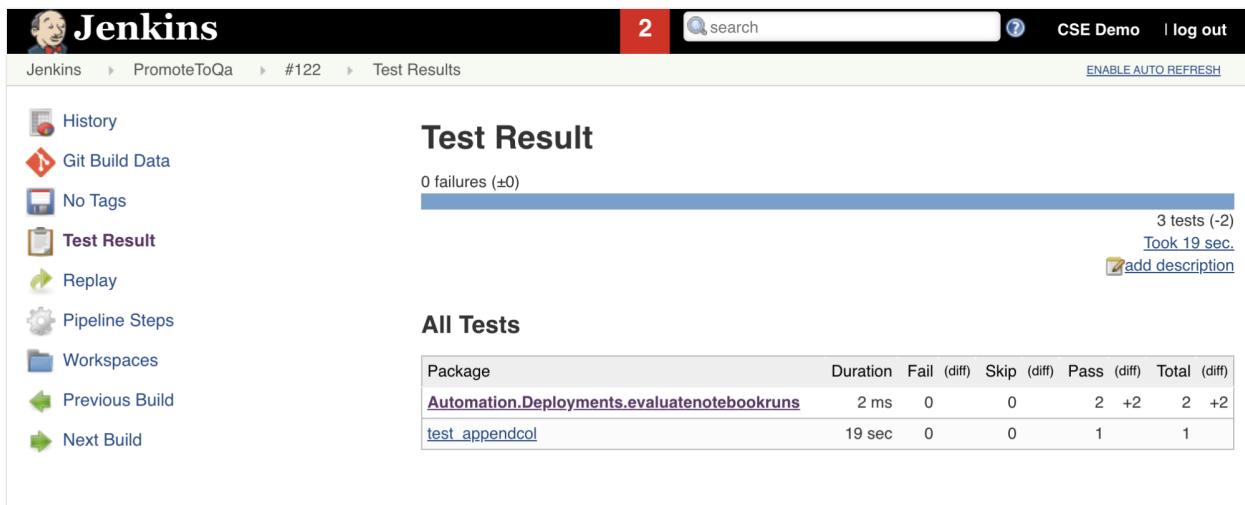
groovy

```
stage('Import Test Results') {
 def DATABRICKS_BUNDLE_WORKSPACE_FILE_PATH
 def getPath = "${DBCLIPATH}/databricks bundle validate -t ${BUNDLETARGET} \
| ${JQPATH}/jq -r .workspace.file_path"
 def output = sh(script: getPath, returnStdout: true).trim()

 if (output) {
 DATABRICKS_BUNDLE_WORKSPACE_FILE_PATH = "${output}"
 } else {
 error "Failed to capture output or command execution failed: ${getPath}"
 }

 sh """
#!/bin/bash
${DBCLIPATH}/databricks workspace export-dir \
${DATABRICKS_BUNDLE_WORKSPACE_FILE_PATH}/Validation/Output/test-
results \
${WORKSPACE}/Validation/Output/test-results \
--overwrite
"""
}

stage('Publish Test Results') {
 junit allowEmptyResults: true, testResults: '**/test-results/*.xml',
 skipPublishingChecks: true
}
```



The screenshot shows the Jenkins Test Results page for a build. The sidebar on the left includes links for History, Git Build Data, No Tags, Test Result (which is selected), Replay, Pipeline Steps, Workspaces, Previous Build, and Next Build. The main content area is titled 'Test Result' and shows '0 failures (±0)'. It indicates '3 tests (-2)' and 'Took 19 sec.'. There is a link to 'add description'. Below this, a table titled 'All Tests' shows the following data:

Package	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total	Total (diff)
Automation.Deployments.evaluateNotebookRuns	2 ms	0	0	2 +2	2	+2
test_appendcol	19 sec	0	0	1	1	

## Push all code changes to your third-party repository

You should now push the contents of your cloned repository on your local development machine to your third-party repository. Before you push, you should first add the following entries to the `.gitignore` file in your cloned repository, as you should probably not push internal Databricks Asset Bundle working files, validation reports, Python build files, and Python caches into your third-party repository. Typically, you will want to regenerate new validation reports and the lastest Python wheel builds in your Azure Databricks workspace, instead of using potentially out-of-date validation reports and Python wheel builds:

```
.databricks/
.vscode/
Libraries/python/dabdemo/build/
Libraries/python/dabdemo/__pycache__/
Libraries/python/dabdemo/dabdemo.egg-info/
Validation/
```

## Run your Jenkins Pipeline

You are now be ready to run your Jenkins Pipeline manually. To do this, from your Jenkins Dashboard:

1. Click the name of your Jenkins Pipeline.
2. On the sidebar, click **Build Now**.
3. To see the results, click the latest Pipeline run (for example, `#1`) and then click **Console Output**.

At this point, the CI/CD pipeline has completed an integration and deployment cycle. By automating this process, you can ensure that your code has been tested and deployed by an efficient, consistent, and repeatable process. To instruct your third-party Git provider to run Jenkins every time a specific event happens, such as a repository pull request, see your third-party Git provider's documentation.

# Power BI security white paper

Article • 12/15/2023

**Summary:** Power BI is an online software service (*SaaS*, or Software as a Service) offering from Microsoft that lets you easily and quickly create self-service Business Intelligence dashboards, reports, semantic models ([previously known as datasets](#)), and visualizations. With Power BI, you can connect to many different data sources, combine and shape data from those connections, then create reports and dashboards that can be shared with others.

**Writers:** Yitzhak Kesselman, Paddy Osborne, Matt Neely, Tony Bencic, Srinivasan Turuvekere, Cristian Petculescu, Adi Regev, Naveen Sivaraj, Ben Glastein, Evgeny Tshiorny, Arthi Ramasubramanian Iyer, Sid Jayadevan, Ronald Chang, Ori Eduar, Anton Fritz, Idan Sheinberg, Ron Gilad, Sagiv Hadaya, Paul Inbar, Igor Uzhviev, Michael Roth, Jaime Tarquino, Gennady Pats, Orion Lee, Yury Berezansky, Maya Shenhav, Romit Chattopadhyay, Yariv Maimon, Bogdan Crivat

**Technical Reviewers:** Cristian Petculescu, Amir Netz, Sergei Gundorov

**Applies to:** Power BI SaaS, Power BI Desktop, Power BI Premium, Power BI Embedded Analytics, Power BI Mobile.

## ⓘ Note

You can save or print this white paper by selecting **Print** from your browser, then selecting **Save as PDF**.

## Introduction

Power BI is an online software service (*SaaS*, or Software as a Service) offering from Microsoft that lets you easily and quickly create self-service Business Intelligence dashboards, reports, semantic models, and visualizations. With Power BI, you can connect to many different data sources, combine and shape data from those connections, then create reports and dashboards that can be shared with others.

The world is rapidly changing; organizations are going through an accelerated digital transformation, and we're seeing a massive increase in remote working, increased customer demand for online services, and increased use of advanced technologies in operations and business decision-making. And all of this is powered by the cloud.

As the transition to the cloud has changed from a trickle to a flood, and with the new, exposed surface area that comes with it, more companies are asking *How secure is my data in the cloud?* and *What end-to-end protection is available to prevent my sensitive data from leaking?* And for the BI platforms that often handle some of the most strategic information in the enterprise, these questions are doubly important.

The decades-old foundations of the BI security model - object-level and row-level security - while still important, clearly no longer suffice for providing the kind of security needed in the cloud era. Instead, organizations must look for a cloud-native, multi-tiered, defense-in-depth security solution for their business intelligence data.

Power BI was built to provide industry-leading complete and hermetic protection for data. The product has earned the highest security classifications available in the industry, and today many national security agencies, financial institutions, and health care providers entrust it with their most sensitive information.

It all starts with the foundation. After a rough period in the early 2000s, Microsoft made massive investments to address its security vulnerabilities, and in the following decades built a strong security stack that goes as deep as the machine on-chip bios kernel and extends all the way up to end-user experiences. These deep investments continue, and today over 3,500 Microsoft engineers are engaged in building and enhancing Microsoft's security stack and proactively addressing the ever-shifting threat landscape. With billions of computers, trillions of logins, and countless zettabytes of information entrusted to Microsoft's protection, the company now possesses the most advanced security stack in the tech industry and is broadly viewed as the global leader in the fight against malicious actors.

Power BI builds on this strong foundation. It uses the same security stack that earned Azure the right to serve and protect the world's most sensitive data, and it integrates with the most advanced information protection and compliance tools of Microsoft 365. On top of these, it delivers security through multi-layered security measures, resulting in end-to-end protection designed to deal with the unique challenges of the cloud era.

To provide an end-to-end solution for protecting sensitive assets, the product team needed to address challenging customer concerns on multiple simultaneous fronts:

- *How do we control who can connect, where they connect from, and how they connect? How can we control the connections?*
- *How is the data stored? How is it encrypted? What controls do I have on my data?*
- *How do I control and protect my sensitive data? How do I ensure this data can't leak outside the organization?*
- *How do I audit who conducts what operations? How do I react quickly if there's suspicious activity on the service?*

This article provides a comprehensive answer to all these questions. It starts with an overview of the service architecture and explains how the main flows in the system work. It then moves on to describe how users authenticate to Power BI, how data connections are established, and how Power BI stores and moves data through the service. The last section discusses the security features that allow you, as the service admin, to protect your most valuable assets.

The Power BI service is governed by the [Microsoft Online Services Terms](#) and the [Microsoft Enterprise Privacy Statement](#). For the location of data processing, refer to the Location of Data Processing terms in the [Microsoft Online Services Terms](#) and to the [Data Protection Addendum](#). For compliance information, the [Microsoft Trust Center](#) is the primary resource for Power BI. The Power BI team is working hard to bring its customers the latest innovations and productivity. Learn more about compliance in the [Microsoft compliance offerings](#).

The Power BI service follows the Security Development Lifecycle (SDL), strict security practices that support security assurance and compliance requirements. The SDL helps developers build more secure software by reducing the number and severity of vulnerabilities in software, while reducing development cost. Learn more at [Microsoft Security Development Lifecycle Practices](#).

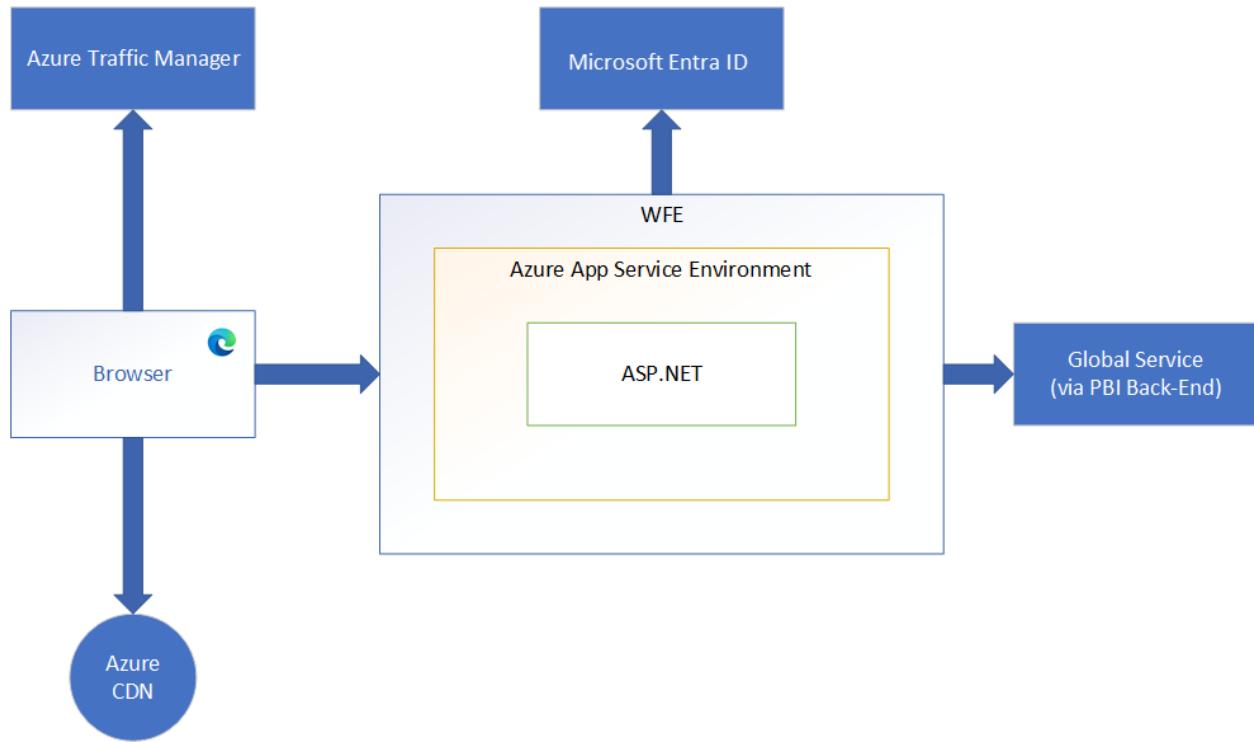
## Power BI architecture

The Power BI service is built on Azure, Microsoft's [cloud computing platform](#). Power BI is currently deployed in many datacenters around the world – there are many active deployments made available to customers in the regions served by those datacenters, and an equal number of passive deployments that serve as backups for each active deployment.



## Web front-end cluster (WFE)

The WFE cluster provides the user's browser with the initial HTML page contents on site load, and pointers to CDN content used to render the site in the browser.



A WFE cluster consists of an ASP.NET website running in the [Azure App Service Environment](#). When users attempt to connect to the Power BI service, the client's DNS service may communicate with the Azure Traffic Manager to find the most appropriate (usually nearest) datacenter with a Power BI deployment. For more information about this process, see [Performance traffic-routing method for Azure Traffic Manager](#).

Static resources such as `*.js`, `*.css`, and image files are mostly stored on an Azure Content Delivery Network (CDN) and retrieved directly by the browser. Note that Sovereign Government cluster deployments are an exception to this rule, and for compliance reasons will omit the CDN and instead use a WFE cluster from a compliant region for hosting static content.

## Power BI back-end cluster (BE)

The back-end cluster is the backbone of all the functionality available in Power BI. It consists of several service endpoints consumed by Web Front End and API clients as well as background working services, databases, caches, and various other components.

The back end is available in most Azure regions, and is being deployed in new regions as they become available. A single Azure region hosts one or more back-end clusters

that allow unlimited horizontal scaling of the Power BI service once the vertical and horizontal scaling limits of a single cluster are exhausted.

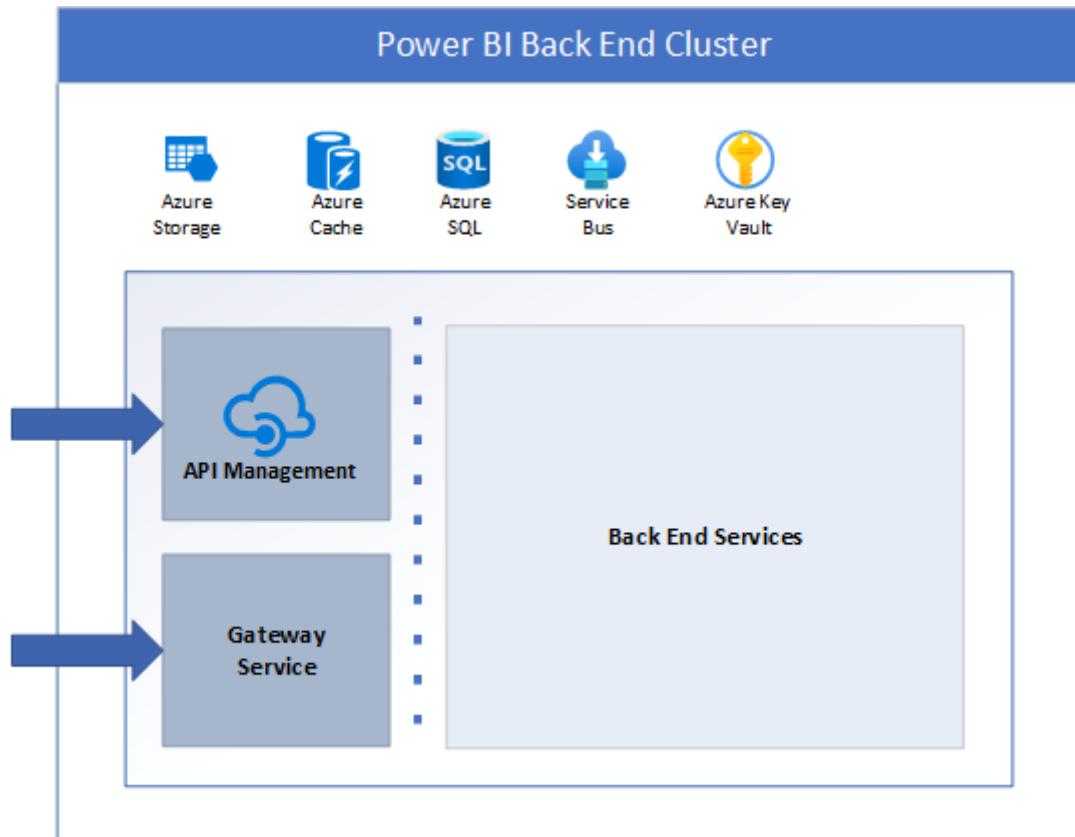
Each back-end cluster is stateful and hosts all the data of all the tenants assigned to that cluster. A cluster that contains the data of a specific tenant is referred to as the tenant's home cluster. An authenticated user's home cluster information is provided by Global Service and used by the Web Front End to route requests to the tenant's home cluster.

Each back-end cluster consists of multiple virtual machines combined into multiple resizable-scale sets tuned for performing specific tasks, stateful resources such as SQL databases, storage accounts, service buses, caches, and other necessary cloud components.

Tenant metadata and data are stored within cluster limits except for data replication to a secondary back-end cluster in a paired Azure region in the same Azure geography. The secondary back-end cluster serves as a failover cluster in case of regional outage, and is passive at any other time.

Back-end functionality is served by micro-services running on different machines within the cluster's virtual network that aren't accessible from the outside, except for two components that can be accessed from the public internet:

- Gateway Service
- Azure API Management

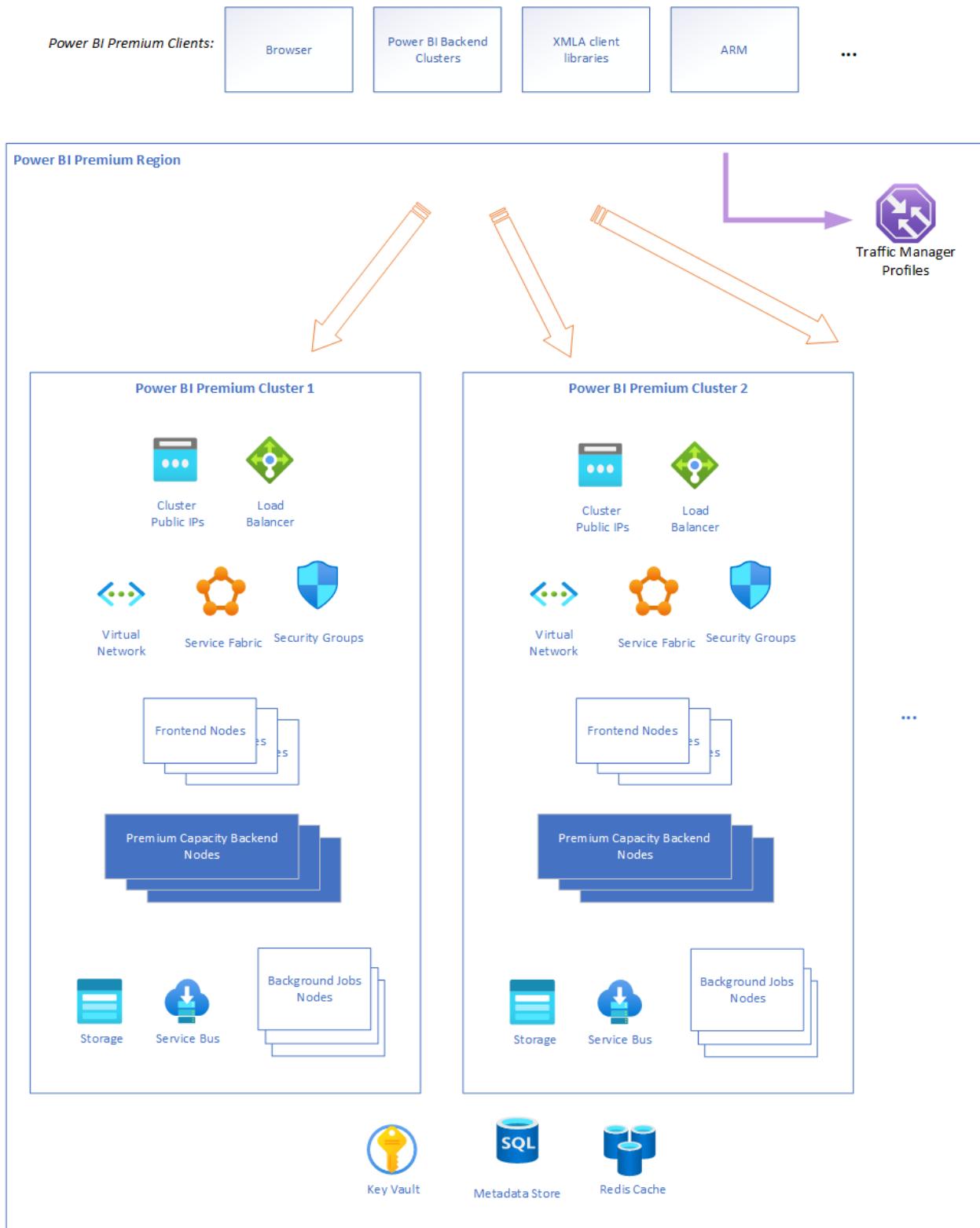


## Power BI Premium infrastructure

Power BI Premium offers a service for subscribers who require premium Power BI features, such as Dataflows, Paginated Reports, AI, etc. When a customer signs up for a Power BI Premium subscription, the Premium capacity is created through the Azure Resource Manager.

Power BI Premium capacities are hosted in back-end clusters that are independent of the regular Power BI back end – see above). This provides better isolation, resource allocation, supportability, security isolation, and scalability of the Premium offering.

The following diagram illustrates the architecture of the Power BI Premium infrastructure:



The connection to the Power BI Premium infrastructure can be done in many ways, depending on the user scenario. Power BI Premium clients can be a user's browser, a regular Power BI back end, direct connections via XMLA clients, ARM APIs, etc.

The Power BI Premium infrastructure in an Azure region consists of multiple Power BI Premium clusters (the minimum is one). Most Premium resources are encapsulated inside a cluster (for instance, compute), and there are some common regional resources (for example, metadata storage). Premium infrastructure allows two ways of achieving

horizontal scalability in a region: increasing resources inside clusters and/or adding more clusters on demand as needed (if cluster resources are approaching their limits).

The backbone of each cluster are compute resources managed by [Virtual Machine Scale Sets](#) and [Azure Service Fabric](#). Virtual Machine Scale Sets and Service Fabric allow fast and painless increase of compute nodes as usage grows and orchestrates the deployment, management, and monitoring of Power BI Premium services and applications.

There are many surrounding resources that ensure a secure and reliable infrastructure: load balancers, virtual networks, network security groups, service bus, storage, etc. Any secrets, keys, and certificates required for Power BI Premium are managed by [Azure Key Vault](#) exclusively. Any authentication is done via integration with Microsoft Entra ID ([previously known as Azure Active Directory](#)) exclusively.

Any request that comes to Power BI Premium infrastructure goes to front-end nodes first – they're the only nodes available for external connections. The rest of the resources are hidden behind virtual networks. The front-end nodes authenticate the request, handle it, or forward it to the appropriate resources (for example, back-end nodes).

Back-end nodes provide most of the Power BI Premium capabilities and features.

## Power BI Mobile

Power BI Mobile is a collection of apps designed for the three primary mobile platforms: Android, iOS, and Windows (UWP). Security considerations for the Power BI Mobile apps fall into two categories:

- Device communication
- The application and data on the device

For device communication, all Power BI Mobile applications communicate with the Power BI service, and use the same connection and authentication sequences used by browsers, which are described in detail earlier in this white paper. The Power BI mobile applications for iOS and Android bring up a browser session within the application itself, while the Windows mobile app brings up a broker to establish the communication channel with Power BI (for the sign-in process).

The following table shows certificate-based authentication (CBA) support for Power BI Mobile, based on the mobile device platform:

[] [Expand table](#)

CBA support	iOS	Android	Windows
Power BI (sign in to service)	Supported	Supported	Not supported
SSRS ADFS on-premises (connect to SSRS server)	Not supported	Supported	Not supported
SSRS App Proxy	Supported	Supported	Not supported

Power BI Mobile apps actively communicate with the Power BI service. Telemetry is used to gather mobile app usage statistics and similar data, which is transmitted to services that are used to monitor usage and activity; no customer data is sent with telemetry.

The Power BI application stores data on the device that facilitates use of the app:

- Microsoft Entra ID and refresh tokens are stored in a secure mechanism on the device, using industry-standard security measures.
- Data and settings (key-value pairs for user configuration) is cached in storage on the device, and can be encrypted by the OS. In iOS this is automatically done when the user sets a passcode. In Android this can be configured in the settings. In Windows it's accomplished by using BitLocker.
- For the Android and iOS apps, the data and settings (key-value pairs for user configuration) are cached in storage on the device in a sandbox and internal storage that is accessible only to the app. For the Windows app, the data is only accessible by the user (and system admin).
- Geolocation is enabled or disabled explicitly by the user. If enabled, geolocation data isn't saved on the device and isn't shared with Microsoft.
- Notifications are enabled or disabled explicitly by the user. If enabled, Android and iOS don't support geographic data residency requirements for notifications.

Data encryption can be enhanced by applying file-level encryption via Microsoft Intune, a software service that provides mobile device and application management. All three platforms for which Power BI Mobile is available support Intune. With Intune enabled and configured, data on the mobile device is encrypted, and the Power BI application itself can't be installed on an SD card. [Learn more about Microsoft Intune](#).

The Windows app also supports [Windows Information Protection \(WIP\)](#).

In order to implement SSO, some secured storage values related to the token-based authentication are available for other Microsoft first party apps (such as Microsoft Authenticator) and are managed by the [Microsoft Authentication Library](#) (MSAL).

Power BI Mobile cached data is deleted when the app is removed, when the user signs out of Power BI Mobile, or when the user fails to sign in (such as after a token expiration

event or password change). The data cache includes dashboards and reports previously accessed from the Power BI Mobile app.

Power BI Mobile doesn't access other application folders or files on the device.

The Power BI apps for iOS and Android let you protect your data by configuring additional identification, such as providing Face ID, Touch ID, or a passcode for iOS, and biometric data (Fingerprint ID) for Android. [Learn more about additional identification](#).

## Authentication to the Power BI service

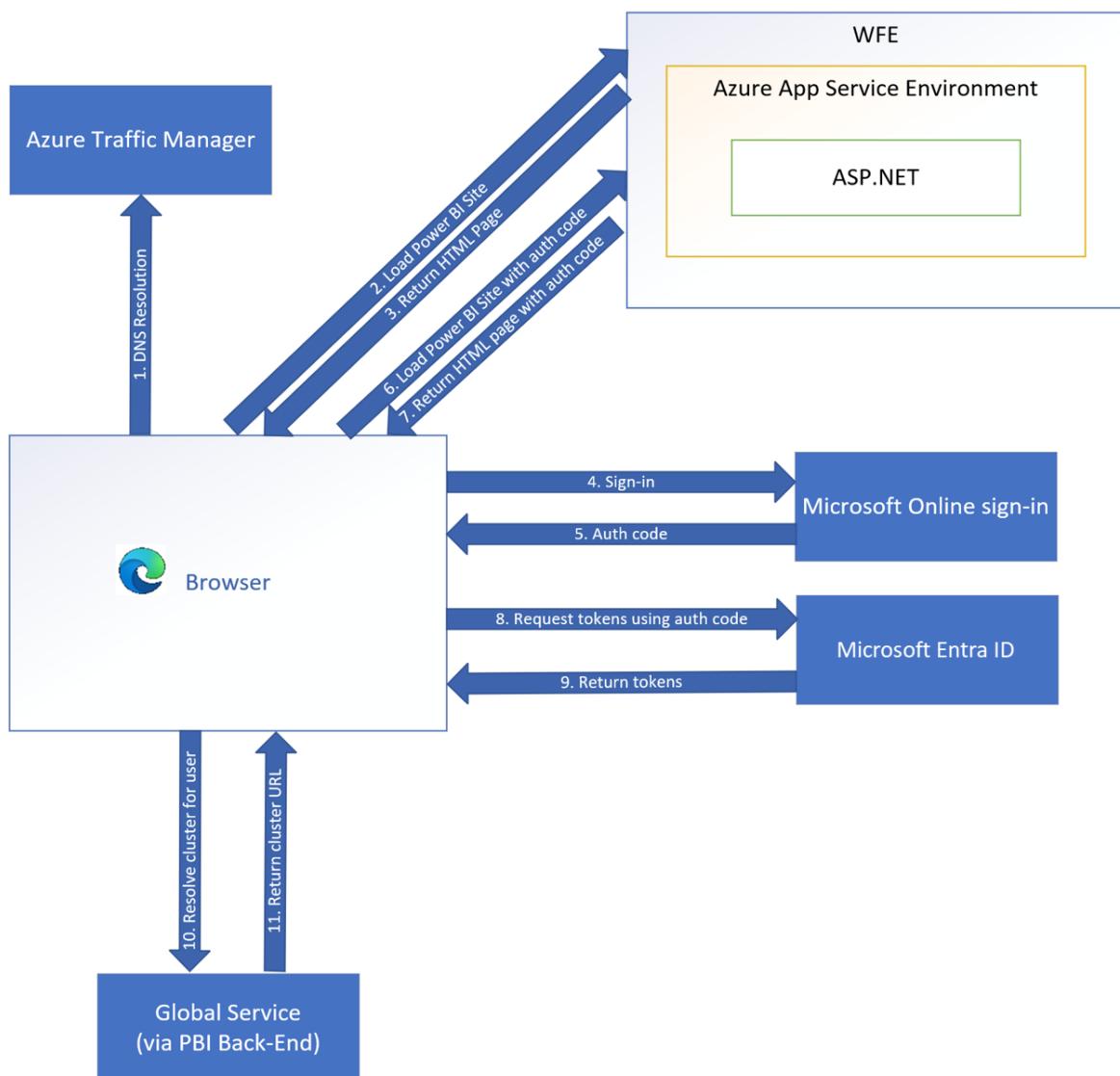
User authentication to the Power BI service consists of a series of requests, responses, and redirects between the user's browser and the Power BI service or the Azure services used by Power BI. That sequence describes the process of user authentication in Power BI, which follows the [Microsoft Entra ID's auth code grant flow](#). For more information about options for an organization's user authentication models (sign-in models), see [Choosing a sign-in model for Microsoft 365](#).

## Authentication sequence

The user authentication sequence for the Power BI service occurs as described in the following steps, which are illustrated in the image that follows them.

1. A user initiates a connection to the Power BI service from a browser, either by typing in the Power BI address in the address bar or by selecting **Sign in** from the Power BI marketing page (<https://powerbi.microsoft.com>). The connection is established using TLS and HTTPS, and all subsequent communication between the browser and the Power BI service uses HTTPS.
2. The Azure Traffic Manager checks the user's DNS record to determine the most appropriate (usually nearest) datacenter where Power BI is deployed, and responds to the DNS with the IP address of the WFE cluster to which the user should be sent.
3. WFE then returns an HTML page to the browser client, which contains a MSAL.js library reference necessary to initiate the sign-in flow.
4. The browser client loads the HTML page received from the WFE, and redirects the user to the Microsoft Online Services sign-in page.
5. After the user has been authenticated, the sign-in page redirects the user back to the Power BI WFE page with an auth code.

6. The browser client loads the HTML page, and uses the auth code to request tokens (access, ID, refresh) from Microsoft Entra ID.
7. The user's tenant ID is used by the browser client to query the Power BI Global Service, which maintains a list of tenants and their Power BI back-end cluster locations. The Power BI Global Service determines which Power BI back-end service cluster contains the user's tenant, and returns the Power BI back-end cluster URL back down to the client.
8. The client is now able to communicate with the Power BI back-end cluster URL API, using the access token in the Authorization header for the HTTP requests. The Microsoft Entra access token will [have an expiry date set according to Microsoft Entra policies](#), and to maintain the current session the Power BI Client in the user's browser will make periodic requests to renew the access token before it expires.



In the rare cases where client-side authentication fails due to an unexpected error, the code attempts to fall back to using server-side authentication in the WFE. Refer to the

questions and answers section at the end of this document for details about the server-side authentication flow.

## Data residency

Unless otherwise indicated in the documentation, Power BI stores customer data in an Azure geography that is assigned when an [Microsoft Entra tenant](#) signs up for Power BI services for the first time. A Microsoft Entra tenant houses the user and application identities, groups, and other relevant information that pertain to an organization and its security.

The assignment of an Azure geography for tenant data storage is done by mapping the country or region selected as part of the Microsoft Entra tenant setup to the most suitable Azure geography where a Power BI deployment exists. Once this determination is made, all Power BI customer data will be stored in this selected Azure geography (also known as the *home geo*), except in cases where organizations utilize multi-geo deployments.

## Multiple geographies (multi-geo)

Some organizations have a global presence and may require Power BI services in multiple Azure geographies. For example, a business may have their headquarters in the United States but may also do business in other geographical areas, such as Australia. In such cases the business may require that certain Power BI data remain stored at rest in the remote region to comply with local regulations. This feature of the Power BI service is referred to as *multi-geo*.

The query execution layer, query caches, and artifact data assigned to a multi-geo workspace are hosted and remain in the remote capacity Azure geography. However, some artifact metadata, such as report structure, may remain stored at rest in the tenant's home geo. Additionally, some data transit and processing may still happen in the tenant's home geo, even for workspaces that are hosted in a multi-geo Premium capacity.

See [Configure Multi-Geo support for Power BI Premium](#) for more information about creating and managing Power BI deployments that span multiple Azure geographies.

## Regions and datacenters

Power BI services are available in specific Azure geographies as described in the [Microsoft Trust Center](#). For more information about where your data is stored and

how it's used, refer to the [Microsoft Trust Center](#). Commitments concerning the location of customer data at rest are specified in the Data Processing Terms of the [Microsoft Online Services Terms](#).

Microsoft also provides datacenters for sovereign entities. For more information about Power BI service availability for national/regional clouds, see [Power BI national/regional clouds](#).

## Data handling

This section outlines Power BI data handling practices when it comes to storing, processing, and transferring customer data.

### Data at rest

Power BI uses two primary data storage resource types:

- Azure Storage
- Azure SQL Databases

In most scenarios, Azure Storage is utilized to persist the data of Power BI artifacts, while Azure SQL Databases are used to persist artifact metadata.

All data persisted by Power BI is encrypted by default using Microsoft-managed keys. Customer data stored in Azure SQL Databases is fully encrypted using [Azure SQL's Transparent Data Encryption \(TDE\)](#) technology. Customer data stored in Azure Blob storage is encrypted using [Azure Storage Encryption](#).

Optionally, organizations can utilize Power BI Premium to use their own keys to encrypt data at rest that is imported into a semantic model. This approach is often described as bring your own key (BYOK). Utilizing BYOK helps ensure that even in case of a service operator error, customer data won't be exposed – something that can't easily be achieved using transparent service-side encryption. See [Bring your own encryption keys for Power BI](#) for more information.

Power BI semantic models allow for various data source connection modes that determine whether the data source data is persisted in the service or not.

Expand table

Semantic Model Mode (Kind)	Data Persisted in Power BI
Import	Yes

Semantic Model Mode (Kind)	Data Persisted in Power BI
DirectQuery	No
Live Connect	No
Composite	If contains an Import data source
Streaming	If configured to persist

Regardless of the semantic model mode utilized, Power BI may temporarily cache any retrieved data to optimize query and report load performance.

## Data in processing

Data is in processing when it's either actively being used by one or more users as part of an interactive scenario, or when a background process, such as refresh, touches this data. Power BI loads actively processed data into the memory space of one or more service workloads. To facilitate the functionality required by the workload, the processed data in memory isn't encrypted.

## Data in transit

Power BI requires all incoming HTTP traffic to be encrypted using TLS 1.2 or above. Any requests attempting to use the service with TLS 1.1 or lower will be rejected.

## Authentication to data sources

When connecting to a data source, a user can choose to import a copy of the data into Power BI or to connect directly to the data source.

In the case of import, a user establishes a connection based on the user's sign in and accesses the data with the credential. After the semantic model is published to the Power BI service, Power BI always uses this user's credential to import data. Once data is imported, viewing the data in reports and dashboards doesn't access the underlying data source. Power BI supports single sign-on authentication for selected data sources. If the connection is configured to use single sign-on, the semantic model owner's credentials are used to connect to the data source.

If a data source is connected directly using preconfigured credentials, the preconfigured credentials are used to connect to the data source when any user views the data. If a data source is connected directly using single sign-on, the current user's credentials are used to connect to the data source when a user views the data. When used with single

sign-on, Row Level Security (RLS) and/or object-level security (OLS) can be implemented on the data source. This allows users to view only data they have privileges to access. When the connection is to data sources in the cloud, Microsoft Entra ID authentication is used for single sign-on; for on-premises data sources, Kerberos, Security Assertion Markup Language (SAML), and Microsoft Entra ID are supported.

If the data source is Azure Analysis Services or on-premises Analysis Services, and RLS and/or OLS is configured, the Power BI service will apply that row level security, and users who don't have sufficient credentials to access the underlying data (which could be a query used in a dashboard, report, or other data artifact) won't see data they don't have sufficient privileges for.

## Premium features

### Dataflows architecture

Dataflows provide users the ability to configure back-end data processing operations that will extract data from polymorphous data sources, execute transformation logic against the data, and then land it in a target model for use across various reporting presentation technologies. Any user who has either a member, contributor, or admin role in a workspace may create a dataflow. Users in the viewer role may view data processed by the dataflow but may not make changes to its composition. Once a dataflow has been authored, any member, contributor, or admin of the workspace may schedule refreshes, as well as view and edit the dataflow by taking ownership of it.

Each configured data source is bound to a client technology for accessing that data source. The structure of credentials required to access them is formed to match required implementation details of the data source. Transformation logic is applied by Power Query services while the data is in flight. For premium dataflows, Power Query services execute in back-end nodes. Data may be pulled directly from the cloud sources or through a gateway installed on premises. When pulled directly from a cloud source to the service or to the gateway, the transport uses protection methodology specific to the client technology, if applicable. When data is transferred from the gateway to the cloud service, it is encrypted. See the [Data in Transit](#) section above.

When customer specified data sources require credentials for access, the owner/creator of the dataflow will provide them during authoring. They're stored using standard product-wide credential storage. See the [Authentication to Data Sources](#) section above. There are various approaches users may configure to optimize data persistence and access. By default, the data is placed in a Power BI owned and protected storage account. Storage encryption is enabled on the Blob storage containers to protect the

data while it is at rest. See the [Data at Rest](#) section below. Users may, however, configure their own storage account associated with their own Azure subscription. When doing so, a Power BI service principal is granted access to that storage account so that it may write the data there during refresh. In this case, the storage resource owner is responsible for configuring encryption on the configured ADLS storage account. Data is always transmitted to Blob storage using encryption.

Since performance when accessing storage accounts may be suboptimal for some data, users also have the option to use a Power BI-hosted compute engine to increase performance. In this case, data is redundantly stored in an SQL database that is available for DirectQuery through access by the back-end Power BI system. Data is always encrypted on the file system. If the user provides a key for encrypting the data stored in the SQL database, that key will be used to doubly encrypt it.

When querying using DirectQuery, the encrypted transport protocol HTTPS is used to access the API. All secondary or indirect use of DirectQuery is controlled by the same access controls previously described. Since dataflows are always bound to a workspace, access to the data is always gated by the user's role in that workspace. A user must have at least read access to be able to query the data via any means.

When Power BI Desktop is used to access data in a dataflow, it must first authenticate the user using Microsoft Entra ID to determine if the user has sufficient rights to view the data. If so, a SaaS key is acquired and used to access storage directly using the encrypted transport protocol HTTPS.

The processing of data throughout the pipeline emits Office 365 auditing events. Some of these events will capture security and privacy-related operations.

## Paginated reports

Paginated reports are designed to be printed or shared. They're called paginated because they're formatted to fit well on a page. They display all the data in a table, even if the table spans multiple pages. You can control their report page layout exactly.

Paginated reports support rich and powerful expressions written in Microsoft Visual Basic .NET. Expressions are widely used throughout Power BI Report Builder paginated reports to retrieve, calculate, display, group, sort, filter, parameterize, and format data.

Expressions are created by the author of the report with access to the broad range of features of the .NET framework. The processing and execution of paginated reports is performed inside a sandbox.

Paginated report definitions (.rdl) are stored in Power BI, and to publish and/or render a paginated report a user needs to authenticate and authorize in the same way as described in the [Authentication to the Power BI Service](#) section above.

The Microsoft Entra token obtained during the authentication is used to communicate directly from the browser to the Power BI Premium cluster.

In Power BI Premium, the Power BI service runtime provides an appropriately isolated execution environment for each report render. This includes cases where the reports being rendered belong to workspaces assigned to the same capacity.

A paginated report can access a wide set of data sources as part of the rendering of the report. The sandbox doesn't communicate directly with any of the data sources but instead communicates with the trusted process to request data, and then the trusted process appends the required credentials to the connection. In this way, the sandbox never has access to any credential or secret.

In order to support features such as Bing maps, or calls to Azure Functions, the sandbox does have access to the internet.

## Power BI embedded analytics

Independent Software Vendors (ISVs) and solution providers have two main modes of embedding Power BI artifacts in their web applications and portals: [embed for your organization](#) and [embed for your customers](#). The artifact is embedded into an IFrame in the application or portal. An IFrame is not allowed to read or write data from the external web application or portal, and the communication with the IFrame is done by using the Power BI Client SDK using POST messages.

In an [embed for your organization](#) scenario, Microsoft Entra users access their own Power BI content through portals customized by their enterprises and ITs. All Power BI policies and capabilities described in this paper such as Row Level Security (RLS) and object-level security (OLS) are automatically applied to all users independently of whether they access Power BI through the [Power BI portal](#) or through customized portals.

In an [embed for your customers](#) scenario, ISVs typically own Power BI tenants and Power BI items (dashboards, reports, semantic models, and others). It's the responsibility of an ISV back-end service to authenticate its end users and decide which artifacts and which access level is appropriate for that end user. ISV policy decisions are encrypted in an [embed token](#) generated by Power BI and passed to the ISV back-end for further distribution to the end users according to the business logic of the ISV. End users using a browser or other client applications aren't able to decrypt or modify embed tokens.

Client-side SDKs such as [Power BI Client APIs](#) automatically append the encrypted embed token to Power BI requests as an *Authorization: EmbedToken* header. Based on this header, Power BI will enforce all policies (such as access or RLS) precisely as was specified by the ISV during generation.

To enable embedding and automation, and to generate the embed tokens described above, Power BI exposes a rich set of [REST APIs](#). These Power BI REST APIs support both user [delegated](#) and [service principal](#) Microsoft Entra methods of authentication and authorization.

Power BI embedded analytics and its REST APIs support all Power BI network isolation capabilities described in this article: For example, [Service Tags](#) and [Private Links](#).

## AI features

Power BI currently supports two broad categories of AI features in the product today: AI visuals and AI enrichments. The visual-level AI features include capabilities such as Key-Influencers, Decomposition-Tree, Smart-Narrative, Anomaly Detection, R-visual, Python-visual, Clustering, Forecasting, Q&A, Quick-Insights etc. The AI enrichment capabilities include capabilities such as AutoML, Azure Machine Learning, CognitiveServices, R/Python transforms etc.

Most of the features mentioned above are supported in both Shared and Premium workspaces today. However, AutoML and CognitiveServices are supported only in Premium workspaces, due to IP restrictions. Today, with the AutoML integration in Power BI, a user can build and train a custom ML model (for example, Prediction, Classification, Regression, etc.) and apply it to get predictions while loading data into a dataflow defined in a Premium workspace. Additionally, Power BI users can apply several CognitiveServices APIs, such as TextAnalytics and ImageTagging, to transform data before loading it into a dataflow/semantic model defined in a Premium workspace.

The Premium AI enrichment features can be best viewed as a collection of stateless AI functions/transforms that can be used by Power BI users in their data integration pipelines used by a Power BI semantic model or dataflow. Note that these functions can also be accessed from current dataflow/semantic model authoring environments in the Power BI Service and Power BI Desktop. These AI functions/transforms always run in a Premium workspace/capacity. These functions are surfaced in Power BI as a data source that requires a Microsoft Entra token for the Power BI user who is using the AI function. These AI data sources are special because they don't surface any of their own data and they only supply these functions/transforms. During execution, these features don't make any outbound calls to other services to transmit the customer's data. Let us look

at the Premium scenarios individually to understand the communication patterns and relevant security-related details pertaining to them.

For training and applying an AutoML model, Power BI uses the Azure AutoML SDK and runs all the training in the customer's Power BI capacity. During training iterations, Power BI calls an experimentation Azure Machine Learning service to select a suitable model and hyper-parameters for the current iteration. In this outbound call, only relevant experiment metadata (for example, accuracy, ml algorithm, algorithm parameters, etc.) from the previous iteration is sent. The AutoML training produces an ONNX model and training report data that is then saved in the dataflow. Later, Power BI users can then apply the trained ML model as a transform to operationalize the ML model on a scheduled basis. For TextAnalytics and ImageTagging APIs, Power BI doesn't directly call the CognitiveServices service APIs, but rather uses an internal SDK to run the APIs in the Power BI Premium capacity. Today these APIs are supported in both Power BI dataflows and semantic models. While authoring a data model in Power BI Desktop, users can only access this functionality if they have access to a Premium Power BI workspace. Hence customers are prompted to supply their Microsoft Entra credentials.

## Network isolation

This section outlines advanced security features in Power BI. Some of the features have specific licensing requirements. See the sections below for details.

## Service tags

A service tag represents a group of IP address prefixes from a given Azure service. It helps minimize the complexity of frequent updates to network security rules. Customers can use service tags to define network access controls on [Network Security Groups](#) or [Azure Firewall](#). Customers can use service tags in place of specific IP addresses when creating security rules. By specifying the service tag name (such as `PowerBI`) in the appropriate source or destination (for APIs) field of a rule, customers can allow or deny the traffic for the corresponding service. Microsoft manages the address prefixes encompassed by the service tag and automatically updates the service tag as addresses change.

## Private Link integration

Azure networking provides the Azure Private Link feature that enables Power BI to provide secure access via Azure Networking private endpoints. With Azure Private Link

and private endpoints, data traffic is sent privately using Microsoft's backbone network infrastructure, and thus the data doesn't traverse the Internet.

Private Link ensures that Power BI users use the Microsoft private network backbone when going to resources in the Power BI service.

Using Private Link with Power BI provides the following benefits:

- Private Link ensures that traffic will flow over the Azure backbone to a private endpoint for Azure cloud-based resources.
- Network traffic isolation from non-Azure-based infrastructure, such as on-premises access, would require customers to have ExpressRoute or a Virtual Private Network (VPN) configured.

See [Private links for accessing Power BI](#) for additional information.

## VNet connectivity (preview - coming soon)

While the Private Link integration feature provides secure inbound connections to Power BI, the VNet connectivity feature enables secure outbound connectivity from Power BI to data sources within a VNet.

VNet gateways (Microsoft-managed) will eliminate the overhead of installing and monitoring on-premises data gateways for connecting to data sources associated with a VNet. They will, however, still follow the familiar process of managing security and data sources, as with an on-premises data gateway.

The following is an overview of what happens when you interact with a Power BI report that is connected to a data source within a VNet using VNet gateways:

1. The Power BI cloud service (or one of the other supported cloud services) kicks off a query and sends the query, data source details, and credentials to the Power Platform VNet service (PP VNet).
2. The PP VNet service then securely injects a container running a VNet gateway into the subnet. This container can now connect to data services accessible from within this subnet.
3. The PP VNet service then sends the query, data source details, and credentials to the VNet gateway.
4. The VNet gateway gets the query and connects to the data sources with those credentials.
5. The query is then sent to the data source for execution.

6. After execution, the results are sent to the VNet gateway, and the PP VNet service securely pushes the data from the container to the Power BI cloud service.

This feature will be available in public preview soon.

## Service principals

Power BI supports the use of service principals. Store any service principal credentials used for encrypting or accessing Power BI in a Key Vault, assign proper access policies to the vault, and regularly review access permissions.

See [Automate Premium workspace and semantic model tasks with service principals](#) for additional details.

## Microsoft Purview for Power BI

### Microsoft Purview Information Protection

Power BI is deeply integrated with Microsoft Purview Information Protection. Microsoft Purview Information Protection enables organizations to have a single, integrated solution for classification, labeling, auditing, and compliance across Azure, Power BI, and Office.

When information protection is enabled in Power BI:

- Sensitive data, both in the Power BI service and in Power BI Desktop, can be classified and labeled using the same sensitivity labels used in Office and in Azure.
- Governance policies can be enforced when Power BI content is exported to Excel, PowerPoint, PDF, Word, or *.pbix* files, to help ensure that data is protected even when it leaves Power BI.
- It's easy to classify and protect *.pbix* files in Power BI Desktop, just like it's done in Excel, Word, and PowerPoint applications. Files can be easily tagged according to their level of sensitivity. Even further, they can be encrypted if they contain business-confidential data, ensuring that only authorized users can edit these files.
- Excel workbooks automatically inherit sensitivity labels when they connect to Power BI (preview), making it possible to maintain end-to-end classification and apply protection when Power BI semantic models are analyzed in Excel.
- Sensitivity labels applied to Power BI reports and dashboards are visible in the Power BI iOS and Android mobile apps.
- Sensitivity labels persist when a Power BI report is embedded in Teams, SharePoint, or a secure website. This helps organizations maintain classification and protection

upon export when embedding Power BI content.

- Label inheritance upon the creation of new content in the Power BI service ensures that labels applied to semantic models or datamarts in the Power BI service will be applied to new content created on top of those semantic models and datamarts.
- [Power BI admin scan APIs](#) can extract a Power BI item's sensitivity label, enabling Power BI and InfoSec admins to monitor labeling in the Power BI service and produce executive reports.
- Power BI admin APIs enable central teams to programmatically apply sensitivity labels to content in the Power BI service.
- Central teams can create mandatory label policies to enforce applying labels on new or edited content in Power BI.
- Central teams can create default label policies to ensure that a sensitivity label is applied to all new or changed Power BI content.
- Automatic downstream sensitivity labeling in the Power BI service ensures that when a label on a semantic model or datamart is applied or changed, the label will automatically be applied or changed on all downstream content connected to the semantic model or datamart.

For more information, see [Sensitivity labels in Power BI](#).

## Microsoft Purview Data Loss Prevention (DLP) Policies for Power BI (preview)

Microsoft Purview's DLP policies can help organizations reduce the risk of sensitive business data leakage from Power BI. DLP policies can help them meet compliance requirements of government or industry regulations, such as GDPR (the European Union's General Data Protection Regulation) or CCPA (the California Consumer Privacy Act) and make sure their data in Power BI is managed.

When DLP policies for Power BI are set up:

- All semantic models within the workspaces specified in the policy are evaluated by the policy.
- You can detect when sensitive data is uploaded into your Premium capacities. DLP policies can detect:
  - Sensitivity labels.
  - Sensitive info types. Over 260 types are supported. Sensitive info type detection relies on Microsoft Purview content scanning.
- When you encounter a semantic model identified as sensitive, you can see a customized policy tip that helps you understand what you should do.

- If you are a semantic model owner, you can override a policy and prevent your semantic model from being identified as "sensitive" if you have a valid reason for doing so.
- If you are a semantic model owner, you can report an issue with a policy if you conclude that a sensitive info type has been falsely identified.
- Automatic risk mitigations, such as alerts to the security admin, can be invoked.

For more information, see [Data loss prevention policies for Power BI](#).

## Microsoft Defender for Cloud Apps for Power BI

Microsoft Defender for Cloud Apps is one of the world's leading cloud access security brokers, named as leader in Gartner's Magic Quadrant for the cloud access security broker (CASP) market. Defender for Cloud Apps is used to secure the use of cloud apps. It enables organizations to monitor and control, in real time, risky Power BI sessions such as user access from unmanaged devices. Security administrators can define policies to control user actions, such as downloading reports with sensitive information.

With Defender for Cloud Apps, organizations can gain the following DLP capabilities:

- Set real-time controls to enforce risky user sessions in Power BI. For example, if a user connects to Power BI from outside of their country or region, the session can be monitored by the Defender for Cloud Apps real-time controls, and risky actions, such as downloading data tagged with a "Highly Confidential" sensitivity label, can be blocked immediately.
- Investigate Power BI user activity with the Defender for Cloud Apps activity log. The Defender for Cloud Apps activity log includes Power BI activity as captured in the Office 365 audit log, which contains information about all user and admin activities, as well as sensitivity label information for relevant activities such as apply, change, and remove label. Admins can leverage the Defender for Cloud Apps advanced filters and quick actions for effective issue investigation.
- Create custom policies to alert on suspicious user activity in Power BI. The Defender for Cloud Apps activity policy feature can be leveraged to define your own custom rules, to help you detect user behavior that deviates from the norm, and even possibly act upon it automatically, if it seems too dangerous.
- Work with the Defender for Cloud Apps built-in anomaly detection. The Defender for Cloud Apps anomaly detection policies provide out-of-the-box user behavioral analytics and machine learning so that you're ready from the outset to run advanced threat detection across your cloud environment. When an anomaly detection policy identifies a suspicious behavior, it triggers a security alert.

- Power BI admin role in the Defender for Cloud Apps portal. Defender for Cloud Apps provides an app-specific admin role that can be used to grant Power BI admins only the permissions they need to access Power BI-relevant data in the portal, such as alerts, users at risk, activity logs, and other Power BI-related information.

See [Using Microsoft Defender for Cloud Apps Controls in Power BI](#) for additional details.

## Preview security features

This section lists features that are planned to release through March 2021. Because this topic lists features that may not have released yet, **delivery timelines may change and projected functionality may be released later than March 2021, or may not be released at all**. For more information, about previews, please review the [Online Services Terms](#).

### Bring Your Own Log Analytics (BYOLA)

Bring Your Own Log Analytics enables integration between Power BI and Azure Log Analytics. This integration includes Azure Log Analytics' advanced analytic engine, interactive query language, and built-in machine learning constructs.

## Power BI security questions and answers

The following questions are common security questions and answers for Power BI. These are organized based on when they were added to this white paper, to facilitate your ability to quickly find new questions and answers when this paper is updated. The newest questions are added to the end of this list.

### How do users connect to, and gain access to data sources while using Power BI?

- Power BI manages credentials to data sources for each user for cloud credentials or for connectivity through a personal gateway. Data sources managed by an on-premises data gateway can be shared across the enterprise and permissions to these data sources can be managed by the Gateway Admin. When configuring a semantic model, the user is allowed to select a credential from their personal store or use an on-premises data gateway to use a shared credential.

In the import case, a user establishes a connection based on the user's sign in and accesses the data with the credential. After the semantic model is published to Power BI service, Power BI always uses this user's credential to import data. Once data is imported, viewing the data in reports and dashboard doesn't access the

underlying data source. Power BI supports single sign-on authentication for selected data sources. If the connection is configured to use single sign-on, the semantic model owner's credential is used to connect with the data source.

For reports that are connected with DirectQuery, the data source is connected directly using a preconfigured credential, the preconfigured credential is used to connect to the data source when any user views the data. If a data source is connected directly using single sign-on, the current user's credential is used to connect to the data source when the user views the data. When using with single sign-on, Row Level Security (RLS) and/or object-level security (OLS) can be implemented on the data source, and this allows users to view data they have privileges to access. When the connection is to data sources in the cloud, Microsoft Entra authentication is used for single sign-on; for on-premises data sources, Kerberos, SAML, and Microsoft Entra ID are supported.

When connecting with Kerberos, the user's UPN is passed to the gateway, and using Kerberos constrained delegation, the user is impersonated and connected to the respective data sources. SAML is also supported on the Gateway for SAP HANA datasource. More information is available in [overview of single sign-on for gateways](#).

If the data source is Azure Analysis Services or on-premises Analysis Services and Row Level Security (RLS) and/or object-level security (OLS) is configured, the Power BI service will apply that row level security, and users who don't have sufficient credentials to access the underlying data (which could be a query used in a dashboard, report, or other data artifact) won't see data for which the user doesn't have sufficient privileges.

[Row Level security with Power BI](#) can be used to restrict data access for given users. Filters restrict data access at the row level, and you can define filters within role.

[Object-level security \(OLS\)](#) can be used to secure sensitive tables or columns. However, unlike row-level security, object-level security also secures object names and metadata. This helps prevent malicious users from discovering even the existence of such objects. Secured tables and columns are obscured in the field list when using reporting tools like Excel or Power BI, and moreover, users without permissions can't access secured metadata objects via DAX or any other method. From the standpoint of users without proper access permissions, secured tables and columns simply don't exist.

Object-level security, together with row-level security, enables enhanced enterprise grade security on reports and semantic models, ensuring that only users with the requisite permissions have access to view and interact with sensitive data.

## How is data transferred to Power BI?

- All data requested and transmitted by Power BI is encrypted in transit using HTTPS (except when the data source chosen by the customer doesn't support HTTPS) to connect from the data source to the Power BI service. A secure connection is established with the data provider, and only once that connection is established will data traverse the network.

## How does Power BI cache report, dashboard, or model data, and is it secure?

- When a data source is accessed, the Power BI service follows the process outlined in the [Authentication to Data Sources](#) section earlier in this document.

## Do clients cache web page data locally?

- When browser clients access Power BI, the Power BI web servers set the *Cache-Control* directive to *no-store*. The *no-store* directive instructs browsers not to cache the web page being viewed by the user, and not to store the web page in the client's cache folder.

## What about role-based security, sharing reports or dashboards, and data connections? How does that work in terms of data access, dashboard viewing, report access or refresh?

- For **non-Role Level Security (RLS)** enabled data sources, if a dashboard, report, or data model is shared with other users through Power BI, the data is then available for users with whom it's shared to view and interact with. Power BI *does not* reauthenticate users against the original source of the data; once data is uploaded into Power BI, the user who authenticated against the source data is responsible for managing which other users and groups can view the data.

When data connections are made to an **RLS**-capable data source, such as an Analysis Services data source, only dashboard data is cached in Power BI. Each time a report or semantic model is viewed or accessed in Power BI that uses data from the RLS-capable data source, the Power BI service accesses the data source to get data based on the user's credentials, and if sufficient permissions exist, the data is loaded into the report or data model for that user. If authentication fails, the user will see an error.

For more information, see the [Authentication to Data Sources](#) section earlier in this document.

Our users connect to the same data sources all the time, some of which require credentials that differ from their domain credentials. How can they avoid having to

input these credentials each time they make a data connection?

- Power BI offers the [Power BI Personal Gateway](#), which is a feature that lets users create credentials for multiple different data sources, then automatically use those credentials when subsequently accessing each of those data sources. For more information, see [Power BI Personal Gateway](#).

Which ports are used by on-premises data gateway and personal gateway? Are there any domain names that need to be allowed for connectivity purposes?

- The detailed answer to this question is available at the following link: [Gateway ports](#)

When working with the on-premises data gateway, how are recovery keys used and where are they stored? What about secure credential management?

- During gateway installation and configuration, the administrator types in a gateway **Recovery Key**. That **Recovery Key** is used to generate a strong AES symmetric key. An RSA asymmetric key is also created at the same time.

Those generated keys (RSA and AES) are stored in a file located on the local machine. That file is also encrypted. The contents of the file can only be decrypted by that particular Windows machine, and only by that particular gateway service account.

When a user enters data source credentials in the Power BI service UI, the credentials are encrypted with the public key in the browser. The gateway decrypts the credentials using the RSA private key and re-encrypts them with an AES symmetric key before the data is stored in the Power BI service. With this process, the Power BI service never has access to the unencrypted data.

Which communication protocols are used by the on-premises data gateway, and how are they secured?

- The gateway supports the following two communications protocols:
  - AMQP 1.0 – TCP + TLS: This protocol requires ports 443, 5671-5672, and 9350-9354 to be open for outgoing communication. This protocol is preferred, since it has lower communication overhead.
  - HTTPS – WebSockets over HTTPS + TLS: This protocol uses port 443 only. The WebSocket is initiated by a single HTTP CONNECT message. Once the channel is established, the communication is essentially TCP+TLS. You can force the gateway to use this protocol by modifying a setting described in the [on-premises gateway article](#).

## What is the role of Azure CDN in Power BI?

- As mentioned previously, Power BI uses the Azure Content Delivery Network (CDN) to efficiently distribute the necessary static content and files to users based on geographical locale. To go into further detail, the Power BI service uses multiple CDNs to efficiently distribute necessary static content and files to users through the public Internet. These static files include product downloads (such as Power BI Desktop, the on-premises data gateway, or Power BI apps from various independent service providers), browser configuration files used to initiate and establish any subsequent connections with the Power BI service, as well as the initial secure Power BI sign-in page.

Based on information provided during an initial connection to the Power BI service, a user's browser contacts the specified Azure CDN (or for some files, the WFE) to download the collection of specified common files necessary to enable the browser's interaction with the Power BI service. The browser page then includes the Microsoft Entra token, session information, the location of the associated back-end cluster, and the collection of files downloaded from the Azure CDN and WFE cluster, for the duration of the Power BI service browser session.

## For Power BI visuals, does Microsoft perform any security or privacy assessment of the custom visual code prior to publishing items to the Gallery?

- No. It is the customer's responsibility to review and determine whether custom visual code should be relied upon. All custom visual code is operated in a sandbox environment, so that any errant code in a custom visual doesn't adversely affect the rest of the Power BI service.

## Are there other Power BI visuals that send information outside the customer network?

- Yes. Bing Maps and ESRI visuals transmit data out of the Power BI service for visuals that use those services.

## For template apps, does Microsoft perform any security or privacy assessment of the template app prior to publishing items to the Gallery?

- No. The app publisher is responsible for the content while it's the customer's responsibility to review and determine whether to trust the template app publisher.

## Are there template apps that can send information outside the customer network?

- Yes. It's the customer's responsibility to review the publisher's privacy policy and determine whether to install the template app on tenant. The publisher is

responsible for informing the customer about the app's behavior and capabilities.

**What about data sovereignty? Can we provision tenants in data centers located in specific geographies, to ensure data doesn't leave the country or region borders?**

- Some customers in certain geographies have an option to create a tenant in a national/regional cloud, where data storage and processing is kept separate from all other datacenters. National/Regional clouds have a slightly different type of security, since a separate data trustee operates the national/regional cloud Power BI service on behalf of Microsoft.

Alternatively, customers can also set up a tenant in a specific region. However, such tenants do not have a separate data trustee from Microsoft. Pricing for national/regional clouds is different from the generally available commercial Power BI service. For more information about Power BI service availability for national/regional clouds, see [Power BI national/regional clouds](#).

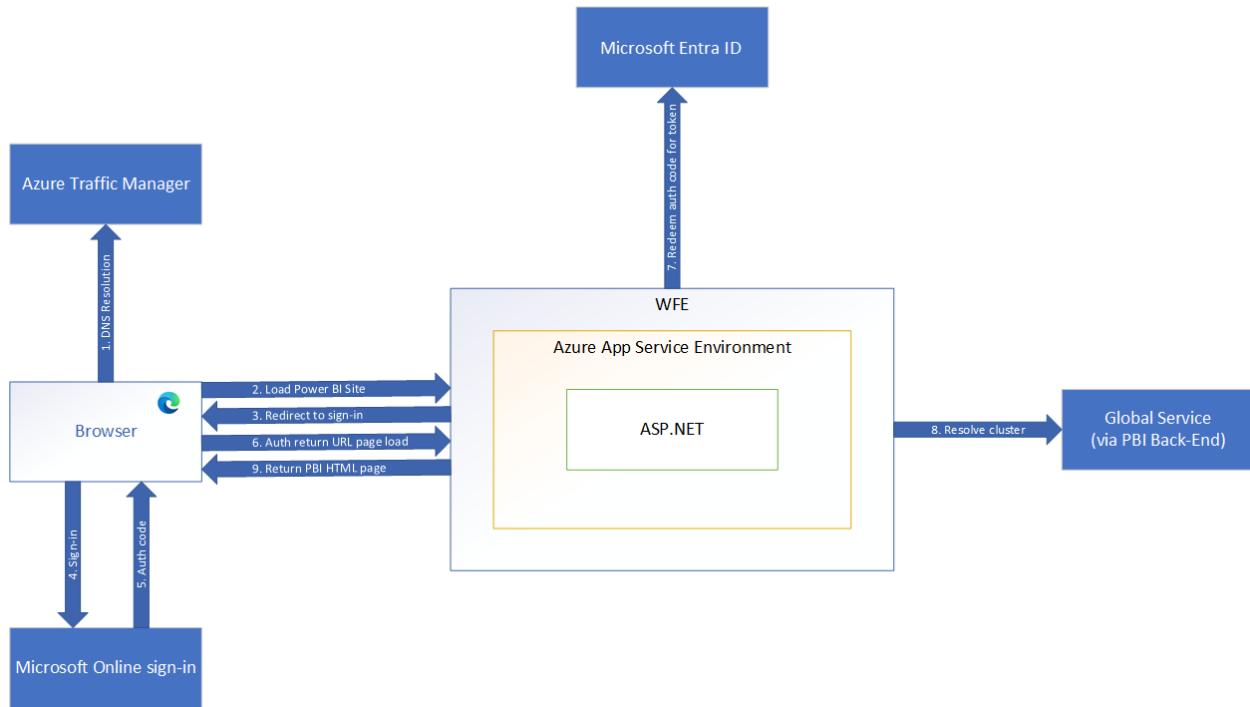
**How does Microsoft treat connections for customers who have Power BI Premium subscriptions? Are those connections different than those established for the non-Premium Power BI service?**

- The connections established for customers with Power BI Premium subscriptions implement a [Microsoft Entra business-to-business \(B2B\)](#) authorization process, using Microsoft Entra ID to enable access control and authorization. Power BI handles connections from Power BI Premium subscribers to Power BI Premium resources just as it would any other Microsoft Entra user.

**How does server-side authentication work in the WFE?**

- The user authentication sequence service-side authentication occurs as described in the following steps, which are illustrated in the image that follows them.
1. A user initiates a connection to the Power BI service from a browser, either by typing in the Power BI address in the address bar or by selecting **Sign in** from the Power BI marketing page (<https://powerbi.microsoft.com>). The connection is established using TLS 1.2 and HTTPS, and all subsequent communication between the browser and the Power BI service uses HTTPS.
  2. The Azure Traffic Manager checks the user's DNS record to determine the most appropriate (usually nearest) datacenter where Power BI is deployed, and responds to the DNS with the IP address of the WFE cluster to which the user should be sent.
  3. WFE then redirects the user to the Microsoft Online Services sign-in page.

4. After the user has been authenticated, the sign-in page redirects the user to the previously determined nearest Power BI service WFE cluster with an auth code.
5. The WFE cluster checks with Microsoft Entra ID to obtain a Microsoft Entra security token by using the auth code. When Microsoft Entra ID returns the successful authentication of the user and returns a Microsoft Entra security token, the WFE cluster consults the Power BI Global Service, which maintains a list of tenants and their Power BI back-end cluster locations and determines which Power BI back-end service cluster contains the user's tenant. The WFE cluster then returns an application page to the user's browser with the session, access, and routing information required for its operation.
6. Now, when the client's browser requires customer data, it will send requests to the back-end cluster address with the Microsoft Entra access token in the Authorization header. The Power BI back-end cluster reads the Microsoft Entra access token and validates the signature to ensure that the identity for the request is valid. The Microsoft Entra access token will [have an expiry date set according to Microsoft Entra policies](#), and to maintain the current session the Power BI Client in the user's browser will make periodic requests to renew the access token before it expires.



## Additional resources

For more information on Power BI, see the following resources.

- [Getting Started with Power BI Desktop](#)
- [Power BI REST API - Overview](#)

- [Power BI API reference](#)
- [On-premises data gateway](#)
- [Power BI National/Regional Clouds ↗](#)
- [Power BI Premium ↗](#)
- [Overview of single sign-on \(SSO\) for gateways in Power BI](#)

# Power BI implementation planning

Article • 01/16/2024

In this video, watch Matthew introduce you to the Power BI implementation planning series of articles.

<https://www.microsoft.com/en-us/videoplayer/embed/RWUWA9?postJs||Msg=true> ↗

Successfully implementing Power BI throughout the organization requires deliberate thought and planning. The Power BI implementation planning series provides you with key considerations, actions, decision-making criteria, and tactical recommendations. The articles in this series cover key subject areas when implementing Power BI, and they describe patterns for common usage scenarios.

## Subject areas

When you implement Power BI, there are many subject areas to consider. The following subject areas form part of the Power BI implementation planning series:

- [BI strategy](#)
- [Tenant setup](#)
- [User tools and devices](#)
- Subscriptions, licenses, and trials
- [Tenant administration](#)
- [Workspaces](#)
- Data management
- Content deployment
- Content distribution and sharing
- [Security](#)
- [Information protection and data loss prevention](#)
- Capacity management
- Data gateways
- Integration with other services
- [Auditing and monitoring](#)
- Adoption tracking
- Scaling and growing

### Note

The series is a work in progress. We will gradually release new and updated articles over time.

## Usage scenarios

The series includes usage scenarios that illustrate different ways that creators and consumers can deploy and use Power BI:

- [Content collaboration and delivery](#)
- [Self-service BI](#)
- [Content management and deployment](#)
- [Embedding and hybrid](#)

## Purpose

When completed, the series will:

- Complement the [Fabric adoption roadmap](#), which describes considerations for successful Microsoft Fabric and Power BI adoption and a healthy data culture. Power BI implementation planning guidance that correlates with the adoption roadmap goals will be added to this series.
- Replace the [Power BI adoption framework](#) (together with the [Fabric adoption roadmap](#)), which is a lightweight set of resources (videos and presentation slides) that were designed to help Microsoft partners deploy Power BI solutions for their customers. Relevant adoption framework action items will be merged into this series.

## Recommendations

To set yourself up for success, we recommend that you work through the following steps:

1. Read the complete [Fabric adoption roadmap](#), familiarizing yourself with each roadmap subject area. Assess your current state of Fabric adoption, and gain clarity on the data culture objectives for your organization.
2. Explore Power BI implementation planning articles that are relevant to you. Start with the [Power BI usage scenarios](#), which convey how you can use Power BI in diverse ways. Be sure to understand which usage scenarios apply to your organization, and by whom. Also, consider how these usage scenarios might influence the implementation strategies you decide on.

3. Read the articles for each of the subject areas that are listed above. You might choose to initially do a broad review of the contents from top to bottom. Or you might choose to start with subject areas that are your highest priority. Carefully review the key decisions and actions that are included for each topic (at the end of each section). We recommend that you use them as a starting point for creating and customizing your plan.

4. When necessary, refer to [Power BI documentation](#) for details on specific topics.

## Target audience

The intended audience of this series of articles might be interested in the following outcomes:

- Identifying areas to improve or strengthen their Power BI implementation.
- Increasing their ability to efficiently manage and securely deliver Power BI content.
- Planning the implementation of Power BI within their organization.
- Increasing their organization's return on investment (ROI) in Power BI.

This series is certain to be helpful for organizations that are in their early stages of a Power BI implementation or are planning an expanded implementation. It might also be helpful for those who work in an organization with one or more of the following characteristics:

- Power BI has pockets of viral adoption and success in the organization, but it's not consistently well-managed or purposefully governed.
- Power BI is deployed with some meaningful scale, but there are many unrealized opportunities for improvement.

### Tip

Some knowledge of Power BI and general business intelligence concepts is assumed. To get the most from this content, we recommend that you become familiar with the [Fabric adoption roadmap](#) first.

## Acknowledgments

The Power BI implementation planning articles are written by Melissa Coates, Kurt Buhler, and Peter Myers. Matthew Roche, from the Fabric Customer Advisory Team, provides strategic guidance and feedback to the subject matter experts.

## Related content

In the [next article in this series](#), learn about usage scenarios that describe how you can use Power BI in diverse ways.

Other helpful resources include:

- [Fabric adoption roadmap](#)
- [Power BI migration overview](#)

Experienced Power BI partners are available to help your organization succeed with the migration process. To engage a Power BI partner, visit the [Power BI partner portal](#) ↗.

# Deploying and Managing Power BI Premium Capacities

Article • 11/10/2023

We have retired the Power BI Premium whitepaper in favor of providing up-to-date information in separate articles. Use the following table to find content from the whitepaper.

Articles	Description
<ul style="list-style-type: none"><li>• <a href="#">Basic concepts for designers in the Power BI service</a></li><li>• <a href="#">Semantic models in the Power BI service</a></li><li>• <a href="#">Semantic model modes in the Power BI service</a></li></ul>	Background information about Power BI service capacities, workspaces, dashboards, reports, workbooks, semantic models, and dataflows.
<ul style="list-style-type: none"><li>• <a href="#">What is Power BI Premium?</a></li></ul>	An overview of Power BI Premium, covering the basics of reserved capacities, supported workloads, unlimited content sharing, and other features.
<ul style="list-style-type: none"><li>• <a href="#">Managing Premium capacities</a></li><li>• <a href="#">Configure and manage capacities in Power BI Premium</a></li><li>• <a href="#">Configure workloads in a Premium capacity</a></li></ul>	Detailed information about configuring and managing capacities and workloads.
<ul style="list-style-type: none"><li>• <a href="#">Use the Microsoft Fabric Capacity Metrics app</a></li></ul>	Monitoring with Power BI Premium Capacity Metrics app, and interpreting the metrics you see in the app.
<ul style="list-style-type: none"><li>• <a href="#">Power BI Premium FAQ</a></li></ul>	Answers to questions around purchase and licensing, features, and common scenarios.

# Introduction to deployment pipelines

Article • 11/15/2023

## ⓘ Note

This articles in this section describe how to deploy content to your app. For version control, see the [Git integration](#) documentation.

In today's world, analytics is a vital part of decision making in almost every organization. Fabric's deployment pipelines tool provides content creators with a production environment where they can collaborate to manage the lifecycle of organizational content. Deployment pipelines enable creators to develop and test content in the service before it reaches the users. See the full list of [Supported item types](#) that you can deploy.

## Learn to use deployment pipelines

You can learn how to use the deployment pipelines tool by following these links.

- [Create and manage a deployment pipeline](#) - A Learn module that walks you through creating a deployment pipeline.
- [Get started with deployment pipelines](#) - An article that explains how to create a pipeline and key functions such as backward deployment and deployment rules.

## Pipeline structure

You can decide how many stages you want in your deployment pipeline. There can be anywhere between two and ten stages. When you create a pipeline, the default three typical stages are given as a starting point, but you can add, delete, or rename the stages to suit your needs. Regardless of how many stages there are, the general concepts are the same.:

- **Development**

The first stage in deployment pipelines where you upload new content with your fellow creators. You can design build, and develop here, or in a different stage.

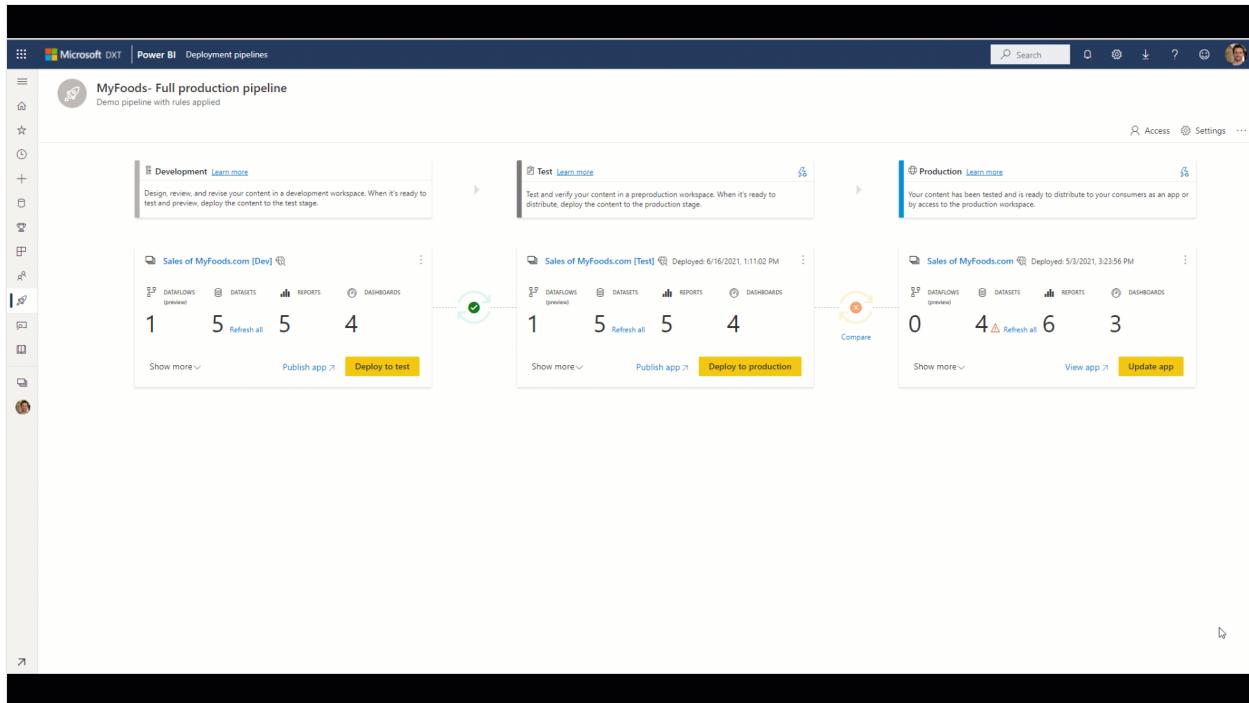
- **Test**

After you've made all the needed changes to your content, you're ready to enter the test stage. Upload the modified content so it can be moved to this test stage. Here are three examples of what can be done in the test environment:

- Share content with testers and reviewers
- Load and run tests with larger volumes of data
- Test your app to see how it will look for your end users

- **Production**

After testing the content, use the production stage to share the final version of your content with business users across the organization.



## Deployment method

When you deploy content from the source stage to a target stage, the source content overwrites anything with the same name in the target stage. Content in the target stage that doesn't exist in the source stage remains in the target stage as is. After you select *deploy*, you'll get a warning message listing the items that will be overwritten.

## Content will be replaced

×

One item in the destination workspace will be affected during deployment

 1 item will be replaced

 Some dataflows are about to be replaced. Replacing dataflows might result in a data loss and would require a full refresh. [Learn more](#)

**Continue**

**Cancel**

You can learn more about [which items are copied to the next stage](#), and [which items are not copied](#), in [Understand the deployment process](#).

## Automation

You can also deploy content programmatically, using the [deployment pipelines REST APIs](#). Learn more about the automation process in [Automate your deployment pipeline using APIs and DevOps](#).

## Related content

- [Understand the deployment pipelines process](#)
- [Get started with deployment pipelines](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# The deployment pipelines process

Article • 01/21/2024

The deployment process lets you clone content from one stage in the deployment pipeline to another, typically from development to test, and from test to production.

During deployment, Microsoft Fabric copies the content from the current stage, into the target one. The connections between the copied items are kept during the copy process. Fabric also applies the configured deployment rules to the updated content in the target stage. Deploying content might take a while, depending on the number of items being deployed. During this time, you can navigate to other pages in the portal, but you can't use the content in the target stage.

You can also deploy content programmatically, using the [deployment pipelines REST APIs](#). You can learn more about this process in [Automate your deployment pipeline using APIs and DevOps](#).

## Deploy content to an empty stage

When you deploy content to an empty stage, a new workspace is created on a capacity for the stage you deploy to. All the metadata in the reports, dashboards, and semantic models of the original workspace is copied to the new workspace in the stage you're deploying to.

There are several ways to deploy content from one stage to another. You can deploy all the content, or you can [select which items to deploy](#).

You can also deploy content backwards, from a later stage in the deployment pipeline, to an earlier one.

After the deployment is complete, refresh the semantic models so that you can use the newly copied content. The semantic model refresh is required because data isn't copied from one stage to another. To understand which item properties are copied during the deployment process, and which item properties aren't copied, review the [item properties copied during deployment](#) section.

## Create a workspace

The first time you deploy content, deployment pipelines checks if you have permissions.

If you have permissions, the content of the workspace is copied to the stage you're deploying to, and a new workspace for that stage is created on the capacity.

If you don't have permissions, the workspace is created but the content isn't copied. You can ask a capacity admin to add your workspace to a capacity, or ask for assignment permissions for the capacity. Later, when the workspace is assigned to a capacity, you can deploy content to this workspace.

If you're using [Premium Per User \(PPU\)](#), your workspace is automatically associated with your PPU. In such cases, permissions aren't required. However, if you create a workspace with a PPU, only other PPU users can access it. In addition, only PPU users can consume content created in such workspaces.

## Workspace and content ownership

The deploying user automatically becomes the owner of the cloned semantic models, and the only admin of the new workspace.

## Deploy content to an existing workspace

Deploying content from a working production pipeline to a stage that has an existing workspace, includes the following steps:

- Deploying new content as an addition to the content already there.
- Deploying updated content to replace some of the content already there.

## Deployment process

When content from the current stage is copied to the target stage, Fabric identifies existing content in the target stage and overwrites it. To identify which content item needs to be overwritten, deployment pipelines uses the connection between the parent item and its clones. This connection is kept when new content is created. The overwrite operation only overwrites the content of the item. The item's ID, URL, and permissions remain unchanged.

In the target stage, [item properties that aren't copied](#), remain as they were before deployment. New content and new items are copied from the current stage to the target stage.

## Auto-binding

In Fabric, when items are connected, one of the items depends on the other. For example, a report always depends on the semantic model it's connected to. A semantic model can depend on another semantic model, and can also be connected to several reports that depend on it. If there's a connection between two items, deployment pipelines always tries to maintain this connection.

During deployment, deployment pipelines checks for dependencies. The deployment either succeeds or fails, depending on the location of the item that provides the data that the deployed item depends on.

- **Linked item exists in the target stage** - Deployment pipelines automatically connect (autobind) the deployed item to the item it depends on in the deployed stage. For example, if you deploy a paginated report from development to test, and it's connected to a semantic model that was previously deployed to the test stage, it automatically connects to that semantic model.
- **Linked item doesn't exist in the target stage** - Deployment pipelines fail a deployment if an item has a dependency on another item, and the item providing the data isn't deployed and doesn't reside in the target stage. For example, if you deploy a report from development to test, and the test stage doesn't contain its semantic model, the deployment will fail. To avoid failed deployments due to dependent items not being deployed, use the *Select related* button. *Select related* automatically selects all the related items that provide dependencies to the items you're about to deploy.

Auto-binding works only with items that are supported by deployment pipelines and reside within Fabric. To view the dependencies of an item, from the item's *More options* menu, select *View lineage*.

The screenshot shows the Power BI service interface. At the top, there is a 'Test' button with a 'Learn more' link. Below it, a message says 'Test and verify your content in a preproduction workspace. When it's ready to distribute, deploy the content to the production stage.' On the right, there is a vertical sidebar with a profile picture and the text 'Your account'.

The main area displays a list of datasets. At the top, there are four summary counts: Dataflows (1), Datasets (2), Reports (4), and Dashboards (2). Below this is a table of datasets:

Dataset	Actions
CDM	View lineage
Marketing analysis	Configure rules
US_Sales_Analysis	⋮
Global sales and marketing	
Marketing analysis	
US_Sales_Analysis	
RegionalSales	
...	

At the bottom of the list are buttons for 'Show less ▾', 'Publish app ↗', and a large yellow 'Deploy to production' button.

A context menu is open for the 'US\_Sales\_Analysis' dataset, listing options: Refresh now, Create report, Delete, Manage permissions, Security, Rename, Settings, and View lineage. The 'View lineage' option is highlighted with a red box.

## Auto-binding across pipelines

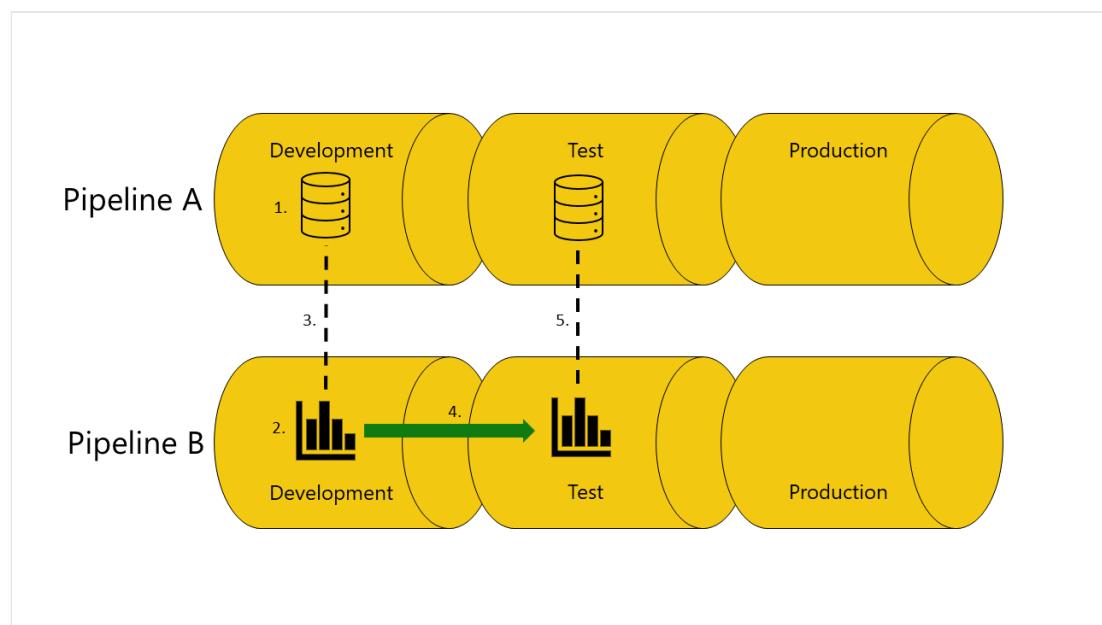
Deployment pipelines automatically binds items that are connected across pipelines, if they're in the same pipeline stage. When you deploy such items, deployment pipelines attempts to establish a new connection between the deployed item and the item it's connected to in the other pipeline. For example, if you have a report in the test stage of pipeline A that's connected to a semantic model in the test stage of pipeline B, deployment pipelines recognizes this connection.

Here's an example with illustrations that to help demonstrate how autobinding across pipelines works:

1. You have a semantic model in the development stage of pipeline A.
2. You also have a report in the development stage of pipeline B.
3. Your report in pipeline B is connected to your semantic model in pipeline A. Your report depends on this semantic model.
4. You deploy the report in pipeline B from the development stage to the test stage.
5. The deployment succeeds or fails, depending on whether or not you have a copy of the semantic model it depends on in the test stage of pipeline A:

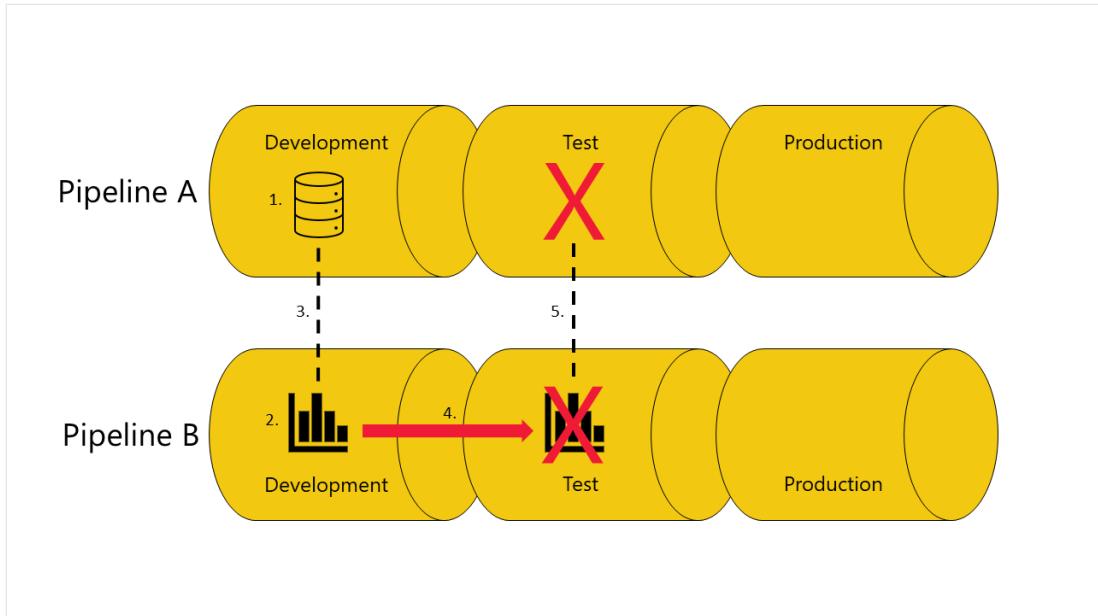
- *If you have a copy of the semantic model the report depends on in the test stage of pipeline A:*

The deployment succeeds, and deployment pipelines connects (auto-bind) the report in the test stage of pipeline B, to the semantic model in the test stage of pipeline A.



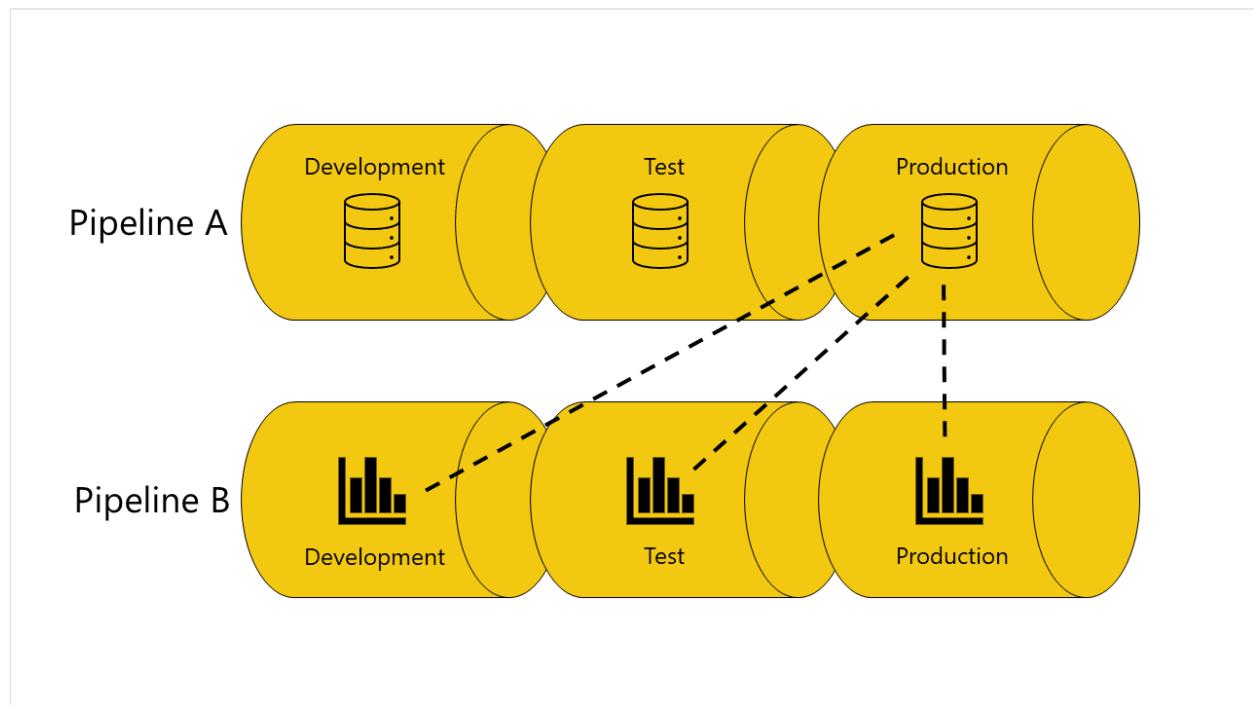
- *If you don't have a copy of the semantic model the report depends on in the test stage of pipeline A:*

The deployment fails because deployment pipelines can't connect (auto-bind) the report in the test stage in pipeline B, to the semantic model it depends on in the test stage of pipeline A.



## Avoid using auto-binding

In some cases, you might not want to use auto-binding. For example, if you have one pipeline for developing organizational semantic models, and another for creating reports. In this case, you might want all the reports to always be connected to semantic models in the production stage of the pipeline they belong to. To accomplish this, avoid using the auto-binding feature.



There are three methods you can use to avoid using auto-binding:

- Don't connect the item to corresponding stages. When the items aren't connected in the same stage, deployment pipelines keeps the original connection. For example, if you have a report in the development stage of pipeline B that's

connected to a semantic model in the production stage of pipeline A. When you deploy the report to the test stage of pipeline B, it remains connected to the semantic model in the production stage of pipeline A.

- Define a parameter rule. This option isn't available for reports. You can only use it with semantic models and dataflows.
- Connect your reports, dashboards, and tiles to a proxy semantic model or dataflow that isn't connected to a pipeline.

## Auto-binding and parameters

Parameters can be used to control the connections between semantic models or dataflows and the items that they depend on. When a parameter controls the connection, autobinding after deployment won't take place, even when the connection includes a parameter that applies to the semantic model's or dataflow's ID, or the workspace ID. In such cases, you'll need to rebind the items after the deployment by changing the parameter value, or by using [parameter rules](#).

### Note

If you're using parameter rules to rebind items, the parameters must be of type [Text](#).

## Refreshing data

Data in the target item, such as a semantic model or dataflow, is kept when possible. If there are no changes to an item that holds the data, the data is kept as it was before the deployment.

In many cases, when you have a small change such as adding or removing a table, Fabric keeps the original data. For breaking schema changes, or changes in the data source connection, a full refresh is required.

## Requirements for deploying to a stage with an existing workspace

Any [licensed user](#) who's a member of both the target and source deployment workspaces, can deploy content that resides on a [capacity](#) to a stage with an existing workspace. For more information, review the [permissions](#) section.

# Supported items

When you deploy content from one pipeline stage to another, the copied content can contain the following items:

- Dataflows Gen1
- Datamarts
- [Lakehouse](#)
- [Notebooks](#)
- Paginated reports
- Reports (based on supported semantic models)
- Semantic models (except for Direct Lake semantic models)
- [Warehouses](#)

## Unsupported items

Deployment pipelines doesn't support the following items:

- Dataflows Gen2
- Data pipelines
- Datasets that don't originate from a *.pbix*
- Direct Lake semantic model
- PUSH datasets
- Streaming dataflows
- Reports based on unsupported semantic models
- [Template app workspaces](#)
- Workbooks
- Metrics

## Item properties copied during deployment

During deployment, the following item properties are copied and overwrite the item properties at the target stage:

- Data sources ([deployment rules](#) are supported)
- Parameters ([deployment rules](#) are supported)
- Report visuals
- Report pages
- Dashboard tiles

- Model metadata
- Item relationships

**Sensitivity labels** are copied *only* when one of the following conditions is met. If these conditions aren't met, sensitivity labels *are not* copied during deployment.

- A new item is deployed, or an existing item is deployed to an empty stage.

 **Note**

In cases where default labeling is enabled on the tenant, and the default label is valid, if the item being deployed is a semantic model or dataflow, the label will be copied from the source item **only** if the label has protection. If the label is not protected, the default label will be applied to the newly created target semantic model or dataflow.

- The source item has a label with protection and the target item doesn't. In such cases, a pop-up window asking for consent to override the target sensitivity label appears.

## Item properties that are not copied

The following item properties aren't copied during deployment:

- Data - Data isn't copied. Only metadata is copied
- URL
- ID
- Permissions - For a workspace or a specific item
- Workspace settings - Each stage has its own workspace
- App content and settings - To update your apps, see [Update content to Power BI apps](#)
- [Personal bookmarks](#)

The following semantic model properties are also not copied during deployment:

- Role assignment
- Refresh schedule

- Data source credentials
- Query caching settings (can be inherited from the capacity)
- Endorsement settings

## Supported semantic model features

Deployment pipelines supports many semantic model features. This section lists two semantic model features that can enhance your deployment pipelines experience:

- [Incremental refresh](#)
- [Composite models](#)

### Incremental refresh

Deployment pipelines supports [incremental refresh](#), a feature that allows large semantic models faster and more reliable refreshes, with lower consumption.

With deployment pipelines, you can make updates to a semantic model with incremental refresh while retaining both data and partitions. When you deploy the semantic model, the policy is copied along.

To understand how incremental refresh behaves with dataflows, see [why do I see two data sources connected to my dataflow after using dataflow rules?](#)

### Activating incremental refresh in a pipeline

To enable incremental refresh, [configure it in Power BI Desktop](#), and then publish your semantic model. After you publish, the incremental refresh policy is similar across the pipeline, and can be authored only in Power BI Desktop.

Once your pipeline is configured with incremental refresh, we recommend that you use the following flow:

1. Make changes to your `.pbix` file in Power BI Desktop. To avoid long waiting times, you can make changes using a sample of your data.
2. Upload your `.pbix` file to the first (usually *development*) stage.
3. Deploy your content to the next stage. After deployment, the changes you made will apply to the entire semantic model you're using.

4. Review the changes you made in each stage, and after you verify them, deploy to the next stage until you get to the final stage.

## Usage examples

The following are a few examples of how you may integrate incremental refresh with deployment pipelines.

- [Create a new pipeline](#) and connect it to a workspace with a semantic model that has incremental refresh enabled.
- Enable incremental refresh in a semantic model that's already in a *development* workspace.
- Create a pipeline from a production workspace that has a semantic model that uses incremental refresh. This is done by using [backwards deployment](#). For example, assign the workspace to a new pipeline's *production* stage, and use backwards deployment to deploy to the *test* stage, and then to the *development* stage.
- Publish a semantic model that uses incremental refresh to a workspace that's part of an existing pipeline.

## Incremental refresh limitations

For incremental refresh, deployment pipelines only supports semantic models that use [enhanced semantic model metadata](#). All semantic models created or modified with Power BI Desktop automatically implement enhanced semantic model metadata.

When republishing a semantic model to an active pipeline with incremental refresh enabled, the following changes result in deployment failure due to data loss potential:

- Republishing a semantic model that doesn't use incremental refresh, to replace a semantic model that has incremental refresh enabled.
- Renaming a table that has incremental refresh enabled.
- Renaming noncalculated columns in a table with incremental refresh enabled.

Other changes such as adding a column, removing a column, and renaming a calculated column, are permitted. However, if the changes affect the display, you need to refresh before the change is visible.

## Composite models

Using [composite models](#) you can set up a report with multiple data connections.

You can use the composite models functionality to connect a Fabric semantic model to an external semantic models such as Azure Analysis Services. For more information, see [Using DirectQuery for Fabric semantic models and Azure Analysis Services](#).

In a deployment pipeline, you can use composite models to connect a semantic model to another Fabric semantic model external to the pipeline.

## Automatic aggregations

[Automatic aggregations](#) are built on top of user defined aggregations and use machine learning to continuously optimize DirectQuery semantic models for maximum report query performance.

Each semantic model keeps its automatic aggregations after deployment. Deployment pipelines doesn't change a semantic model's automatic aggregation. This means that if you deploy a semantic model with an automatic aggregation, the automatic aggregation in the target stage remains as is, and isn't overwritten by the automatic aggregation deployed from the source stage.

To enable automatic aggregations, follow the instructions in [configure the automatic aggregation](#).

## Hybrid tables

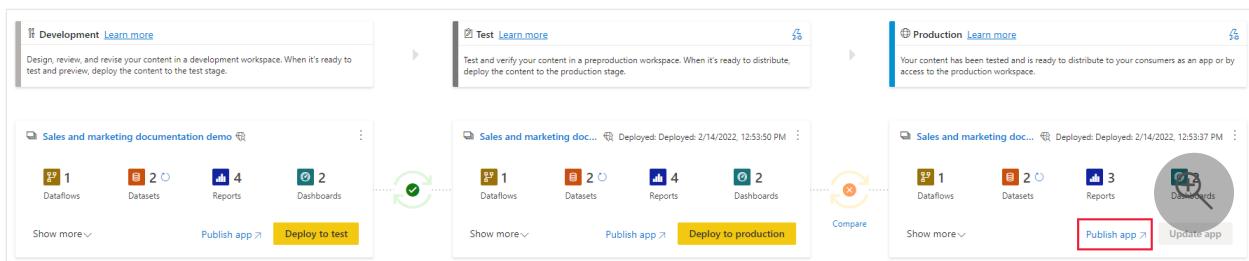
Hybrid tables are tables with [incremental refresh](#) that can have both import and direct query partitions. During a clean deployment, both the refresh policy and the hybrid table partitions are copied. When you're deploying to a pipeline stage that already has hybrid table partitions, only the refresh policy is copied. To update the partitions, refresh the table.

## Update content to Power BI apps

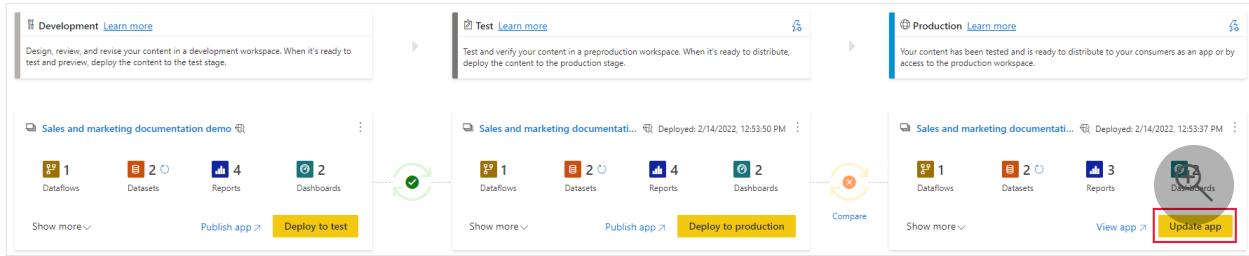
[Power BI apps](#) are the recommended way of distributing content to free Fabric consumers. You can update the content of your Power BI apps using a deployment pipeline, giving you more control and flexibility when it comes to your app's lifecycle.

Create an app for each deployment pipeline stage, so that you can test each update from an end user's point of view. Use the [publish](#) or [view](#) button in the workspace card

to publish or view the app in a specific pipeline stage.



In the production stage, the main action button on the bottom-right corner opens the update app page in Fabric, so that any content updates become available to app users.



## Important

The deployment process does not include updating the app content or settings. To apply changes to content or settings, you need to manually update the app in the required pipeline stage.

## Permissions

Permissions are required for the pipeline, and for the workspaces that are assigned to it. Pipeline permissions and workspace permissions are granted and managed separately.

- Pipelines only have one permission, *Admin*, which is required for sharing, editing and deleting a pipeline.
- Workspaces have different permissions, also called **roles**. Workspace roles determine the level of access to a workspace in a pipeline.
- Deployment pipelines do not support **Microsoft 365 groups** as pipeline admins.

To deploy from one stage to another in the pipeline, you must be a pipeline admin, and either a member or an admin of the workspaces assigned to the stages involved. For example, a pipeline admin that isn't assigned a workspace role, will be able to view the pipeline and share it with others. However, this user won't be able to view the content of the workspace in the pipeline, or in the service, and won't be able to perform deployments.

# Permissions table

This section describes the deployment pipeline permissions. The permissions listed in this section may have different applications in other Fabric features.

The lowest deployment pipeline permission is *pipeline admin*, and it's required for all deployment pipeline operations.

 [Expand table](#)

User	Pipeline permissions	Comments
<b>Pipeline admin</b>	<ul style="list-style-type: none"><li>View the pipeline</li><li>Share the pipeline with others</li><li>Edit and delete the pipeline</li><li>Unassign a workspace from a stage</li><li>Can see workspaces that are tagged as assigned to the pipeline in Power BI service</li></ul>	Pipeline access doesn't grant permissions to view or take actions on the workspace content.
<b>Workspace viewer (and pipeline admin)</b>	<ul style="list-style-type: none"><li>Consume content</li><li>Unassign a workspace from a stage</li></ul>	Workspace members assigned the Viewer role without <i>build</i> permissions, can't access the semantic model or edit workspace content.
<b>Workspace contributor (and pipeline admin)</b>	<ul style="list-style-type: none"><li>Consume content</li><li>Compare stages</li><li>View semantic models</li><li>Unassign a workspace from a stage</li></ul>	
<b>Workspace member (and pipeline admin)</b>	<ul style="list-style-type: none"><li>View workspace content</li><li>Compare stages</li><li>Deploy items (must be a member or admin of both source and target workspaces)</li><li>Update semantic models</li><li>Unassign a workspace from a stage</li><li>Configure semantic model rules (you must be</li></ul>	If the <i>block republish and disable package refresh</i> setting located in the tenant <i>semantic model security</i> section is enabled, only semantic model owners are able to update semantic models.

User	Pipeline permissions	Comments
	the semantic model owner)	
<b>Workspace admin (and pipeline admin)</b>	<ul style="list-style-type: none"> <li>• View workspace content</li> <li>• Compare stages</li> <li>• Deploy items</li> <li>• Assign workspaces to a stage</li> <li>• Update semantic models</li> <li>• Unassign a workspace from a stage</li> <li>• Configure semantic model rules (you must be the semantic model owner)</li> </ul>	

## Granted permissions

When you're deploying Power BI items, the ownership of the deployed item may change. Review the following table to understand who can deploy each item and how the deployment affects the item's ownership.

[Expand table](#)

Fabric Item	Required permission to deploy an existing item	Item ownership after a first time deployment	Item ownership after deployment to a stage with the item
Semantic model	Workspace member	The user who made the deployment becomes the owner	Unchanged
Dataflow	Dataflow owner	The user who made the deployment becomes the owner	Unchanged
Datamart	Datamart owner	The user who made the deployment becomes the owner	Unchanged
Paginated report	Workspace member	The user who made the deployment becomes the owner	The user who made the deployment becomes the owner

# Required permissions for popular actions

The following table lists required permissions for popular deployment pipeline actions. Unless specified otherwise, for each action you need all the listed permissions.

 Expand table

Action	Required permissions
View the list of pipelines in your organization	No license required (free user)
Create a pipeline	A user with one of the following licenses: <ul style="list-style-type: none"><li>• Pro</li><li>• PPU</li><li>• Premium</li></ul>
Delete a pipeline	Pipeline admin
Add or remove a pipeline user	Pipeline admin
Assign a workspace to a stage	<ul style="list-style-type: none"><li>• Pipeline admin</li><li>• Workspace admin (of the workspace to be assigned)</li></ul>
Unassign a workspace to a stage	One of the following: <ul style="list-style-type: none"><li>• Pipeline admin</li><li>• Workspace admin (using the <a href="#">Pipelines - Unassign Workspace API</a>)</li></ul>
Deploy to an empty stage	<ul style="list-style-type: none"><li>• Pipeline admin</li><li>• Source workspace member or admin</li></ul>
Deploy items to the next stage	<ul style="list-style-type: none"><li>• Pipeline admin</li><li>• Workspace member or admin of both the source and target stages</li><li>• To deploy datamarts or dataflows, you must be the owner of the deployed item</li><li>• If the semantic model tenant admin switch is turned on and you're deploying a semantic model, you need to be the owner of the semantic model</li></ul>
View or set a rule	<ul style="list-style-type: none"><li>• Pipeline admin</li><li>• Target workspace contributor, member, or admin</li><li>• Owner of the item you're setting a rule for</li></ul>

Action	Required permissions
Manage pipeline settings	Pipeline admin
View a pipeline stage	<ul style="list-style-type: none"> <li>• Pipeline admin</li> <li>• Workspace reader, contributor, member, or admin. You'll see the items that your workspace permissions grant access to.</li> </ul>
View the list of items in a stage	Pipeline admin
Compare two stages	<ul style="list-style-type: none"> <li>• Pipeline admin</li> <li>• Workspace contributor, member, or admin for both stages</li> </ul>
View deployment history	Pipeline admin

## Considerations and limitations

This section lists most of the limitations in deployment pipelines.

- The workspace must reside on a [Fabric capacity](#).
- The maximum number of items that can be deployed in a single deployment is 300.
- Downloading a *.pbix* file after deployment isn't supported.
- [Microsoft 365 groups](#) aren't supported as pipeline admins.
- When you're deploying a Power BI item for the first time, if another item in the target stage is similar in type (for example, if both files are reports) and has the same name, the deployment fails.
- For a list of workspace limitations, see the [workspace assignment limitations](#).
- For a list of unsupported items, see [unsupported items](#).
- The deployment fails if any of the items have circular or self dependencies (for example, item A references item B and item B references item A).

## Semantic model limitations

- Datasets that use real-time data connectivity can't be deployed.

- A semantic model with DirectQuery or Composite connectivity mode that uses variation or [auto date/time](#) tables, isn't supported. For more information, see [What can I do if I have a dataset with DirectQuery or Composite connectivity mode, that uses variation or calendar tables?](#).
- During deployment, if the target semantic model is using a [live connection](#), the source semantic model must use this connection mode too.
- After deployment, downloading a semantic model (from the stage it's been deployed to) isn't supported.
- For a list of deployment rule limitations, see [deployment rules limitations](#).
- Deployment is **not** supported on a semantic model that uses Native query and DirectQuery together and auto binding is engaged on the DirectQuery data source.

## Dataflow limitations

- When you're deploying a dataflow to an empty stage, deployment pipelines creates a new workspace and sets the dataflow storage to a Fabric blob storage. Blob storage is used even if the source workspace is configured to use Azure data lake storage Gen2 (ADLS Gen2).
- Service principal isn't supported for dataflows.
- Deploying common data model (CDM) isn't supported.
- For deployment pipeline rule limitations that affect dataflows, see [Deployment rules limitations](#).
- If a dataflow is being refreshed during deployment, the deployment fails.
- When comparing stages during dataflow refresh, the results are unpredictable.

## Datamart limitations

- You can't deploy a datamart with sensitivity labels.
- To deploy a datamart, you need to be the datamart owner.

## Related content

[Get started with deployment pipelines](#)

---

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Lifecycle management best practices

Article • 11/15/2023

This article provides guidance for data & analytics creators who are managing their content throughout its lifecycle in Microsoft Fabric. The article focuses on the use of [Git integration](#) for source control and [deployment pipelines](#) as a release tool. For a general guidance on Enterprise content publishing, [Enterprise content publishing](#).

## ⓘ Important

This feature is in [preview](#).

The article is divided into four sections:

- [Content preparation](#) - Prepare your content for lifecycle management.
- [Development](#) - Learn about the best ways of creating content in the deployment pipelines development stage.
- [Test](#) - Understand how to use a deployment pipelines test stage to test your environment.
- [Production](#) - Utilize a deployment pipelines production stage to make your content available for consumption.

## Content preparation

To best prepare your content for on-going management throughout its lifecycle, review the information in this section before you:

- Release content to production.
- Start using a deployment pipeline for a specific workspace.

## Separate development between teams

Different teams in the org usually have different expertise, ownership, and methods of work, even when working on the same project. It's important to set boundaries while giving each team their independence to work as they like. Consider having separate workspaces for different teams. With separate workspaces, each team can have different permissions, work with different source control repos, and ship content to production in

a different cadence. Most items can connect and use data across workspaces, so it doesn't block collaboration on the same data and project.

## Plan your permission model

Both Git integration and deployment pipelines require different permissions than just the workspace permissions. Read about the permission requirements for [Git integration and deployment pipelines](#).

To implement a secure and easy workflow, plan who gets access to each part of the environments being used, both the Git repository and the dev/test/prod stages in a pipeline. Some of the considerations to take into account are:

- Who should have access to the source code in the Git repository?
- Which operations should users with pipeline access be able to perform in each stage?
- Who's reviewing content in the test stage?
- Should the test stage reviewers have access to the pipeline?
- Who should oversee deployment to the production stage?
- Which workspace are you assigning to a pipeline, or connecting to git?
- Which branch are you connecting the workspace to? What's the policy defined for that branch?
- Is the workspace shared by multiple team members? Should they make changes directly in the workspace, or only through Pull requests?
- Which stage are you assigning your workspace to?
- Do you need to make changes to the permissions of the workspace you're assigning?

## Connect different stages to different databases

A production database should always be stable and available. It's best not to overload it with queries generated by BI creators for their development or test semantic models. Build separate databases for development and testing in order to protect production data and not overload the development database with the entire volume of production data.

# Use parameters for configurations that will change between stages

Whenever possible, add [parameters](#) to any definition that might change between dev/test/prod stages. Using parameters helps you change the definitions easily when you move your changes to production. While there's still no unified way to manage parameters in Fabric, we recommend using it on items that support any type of parameterization. Parameters have different uses, such as defining connections to data sources, or to internal items in Fabric. They can also be used to make changes to queries, filters, and the text displayed to users.

In deployment pipelines, you can configure parameter rules to set different values for each deployment stage.

## Development

This section provides guidance for working with the deployment pipelines and using fit for your development stage.

### Back up your work into a Git repository

With Git integration, any developer can back up their work by committing it into Git. To back up your work properly in Fabric, here are some basic rules:

- Make sure you have an isolated environment to work in, so others don't override your work before it gets committed. This means working in a Desktop tool (such as [VS Code](#) ↗, [Power BI Desktop](#) ↗ or others), or in a separate workspace that other users can't access.
- Commit to a branch that you created and no other developer is using. If you're using a workspace as an authoring environment, read about [working with branches](#).
- Commit together changes that must be deployed together. This advice applies for a single item, or multiple items that are related to the same change. Committing all related changes together can help you later when deploying to other stages, creating pull requests, or reverting changes back.
- Big commits might hit a max commit size limit. Be mindful of the number of items you commit together, or the general size of an item. For example, reports can grow large when adding large images. It's bad practice to store large-size items in

source control systems, even if it works. Consider ways to reduce the size of your items if they have lots of static resources, like images.

## Rolling back changes

After you back up your work, there might be cases where you want to revert to a previous version and restore it in the workspace. There are a few ways to revert to a previous version:

- **Undo button:** The *Undo* operation is an easy and fast way to revert the immediate changes you made, as long as they aren't committed yet. You can also undo each item separately. Read more about the [undo](#) operation.
- **Reverting to older commits:** There's no direct way to go back to a previous commit in the UI. The best option is to promote an older commit to be the HEAD using [git revert](#) or [git reset](#). Doing this shows that there's an update in the source control pane, and you can update the workspace with that new commit.

Since data isn't stored in Git, keep in mind that reverting a data item to an older version might break the existing data and could possibly require you to drop the data or the operation might fail. Check this in advance before reverting changes back.

## Working with a 'private' workspace

When you want to work in isolation, use a separate workspace as an isolated environment. Read more about isolating your work environment in [working with branches](#). For an optimal workflow for you and the team, consider the following points:

- **Setting up the workspace:** Before you start, make sure you can create a new workspace (if you don't already have one), that you can assign it to a [Fabric capacity](#), and that you have access to data to work in your workspace.
- **Creating a new branch:** Create a new branch from the *main* branch, so you have the most up-to-date version of your content. Also make sure you connect to the correct folder in the branch, so you can pull the right content into the workspace.
- **Small, frequent changes:** It's a Git best practice to make small incremental changes that are easy to merge and less likely to get into conflicts. If that's not possible, make sure to update your branch from main so you can resolve conflicts on your own first.
- **Configuration changes:** If necessary, change the configurations in your workspace to help you work more productively. Some changes can include connection

between items, or to different data sources or changes to parameters on a given item. Just remember that anything you commit becomes part of the commit and can accidentally be merged into the main branch.

## Use Client tools to edit your work

For items and tools that support it, it might be easier to work with client tools for authoring, such as [Power BI Desktop](#) for semantic models and reports, [VSCode](#) for Notebooks etc. These tools can be your local development environment. After you complete your work, push the changes into the remote repo, and sync the workspace to upload the changes. Just make sure you're working with the [supported structure](#) of the item you're authoring. If you're not sure, first clone a repo with content already synced to a workspace, then start authoring from there, where the structure is already in place.

## Managing workspaces and branches

Since a workspace can only be connected to a single branch at a time, it's recommended to treat this as a 1:1 mapping. However, to reduce the amount of workspace it entails, consider these options:

- If a developer set up a private workspace with all required configurations, they can continue to use that workspace for any future branch they create. When a sprint is over, your changes are merged and you're starting a fresh new task, just switch the connection to a new branch on the same workspace. You can also do this if you suddenly need to fix a bug in the middle of a sprint. Think of it as a working directory on the web.
- Developers using a client tool (such as VS Code, Power BI Desktop, or others), don't necessarily need a workspace. They can create branches and commit changes to that branch locally, push those to the remote repo and create a pull request to the main branch, all without a workspace. A workspace is needed only as a testing environment to check that everything works in a real-life scenario. It's up to you to decide when that should happen.

## Duplicate an item in a Git repository

To duplicate an item in a Git repository:

1. Copy the entire item directory.
2. Change the logicalId to something unique for that connected workspace.

3. Change the display name to differentiate it from the original item and to avoid duplicate display name error.
4. If necessary, update the logicalId and/or display names in any dependencies.

## Test

This section provides guidance for working with a deployment pipelines test stage.

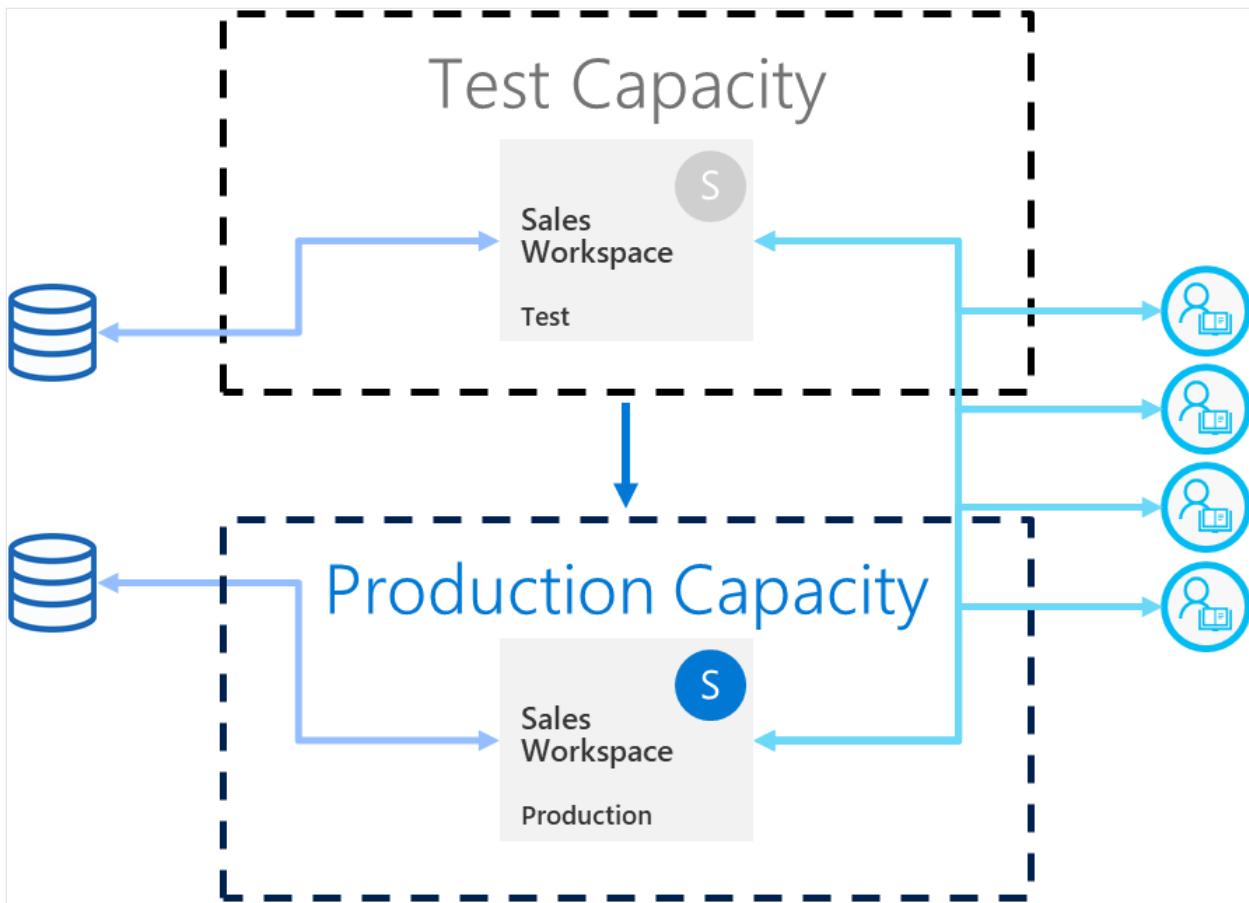
### Simulate your production environment

It's important to see how your proposed change will impact the production stage. A deployment pipelines test stage allows you to simulate a real production environment for testing purposes. Alternatively, you can simulate this by connecting Git to another workspace.

Make sure that these three factors are addressed in your test environment:

- Data volume
- Usage volume
- A similar capacity as in production

When testing, you can use the same capacity as the production stage. However, using the same capacity can make production unstable during load testing. To avoid unstable production, test using a different capacity similar in resources to the production capacity. To avoid extra costs, use a capacity where you can pay only for the testing time.



## Use deployment rules with a real-life data source

If you're using the test stage to simulate real life data usage, it's best to separate the development and test data sources. The development database should be relatively small, and the test database should be as similar as possible to the production database. Use [data source rules](#) to switch data sources in the test stage or parameterize the connection if not working through deployment pipelines.

## Check related items

Changes you make can also affect the dependent items. During testing, verify that your changes don't affect or break the performance of existing items, which can be dependent on the updated ones.

You can easily find the related items by using [impact analysis](#).

## Updating data items

Data items are items that store data. The item's definition in Git defines how the data is stored. When updating an item in the workspace, we're importing its definition into the workspace and applying it on the existing data. The operation of updating data items is the same for Git and deployment pipelines.

As different items have different capabilities when it comes to retaining data when changes to the definition are applied, be mindful when applying the changes. Some practices that can help you apply the changes in the safest way:

- Know in advance what the changes are and what their impact might be on the existing data. Use commit messages to describe the changes made.
- To see how that item handles the change with test data, upload the changes first to a dev or test environment.
- If everything goes well, it's recommended to also check it on a staging environment, with real-life data (or as close to it as possible), to minimize the unexpected behaviors in production.
- Consider the best timing when updating the Prod environment to minimize the damage that any errors might cause to your business users who consume the data.
- After deployment, post-deployment tests in Prod to verify that everything is working as expected.
- Some changes will always be considered *breaking changes*. Hopefully, the preceding steps will help you track them before production. Build a plan for how to apply the changes in Prod and recover the data to get back to normal state and minimize downtime for business users.

## Test your app

If you're distributing content to your customers through an app, review the app's new version *before* it's in production. Since each deployment pipeline stage has its own workspace, you can easily publish and update apps for development and test stages. Publishing and updating apps allows you to test the app from an end user's point of view.

### **Important**

The deployment process doesn't include updating the app content or settings. To apply changes to content or settings, manually update the app in the required pipeline stage.

## Production

This section provides guidance to the deployment pipelines production stage.

## Manage who can deploy to production

Because deploying to production should be handled carefully, it's good practice to let only specific people manage this sensitive operation. However, you probably want all BI creators for a specific workspace to have access to the pipeline. Use production [workspace permissions](#) to manage access permissions. Other users can have a production workspace *viewer* role to see content in the workspace but not make changes from Git or deployment pipelines.

In addition, limit access to the repo or pipeline by only enabling permissions to users that are part of the content creation process.

## Set rules to ensure production stage availability

[Deployment rules](#) are a powerful way to ensure the data in production is always connected and available to users. With deployment rules applied, deployments can run while you have the assurance that customers can see the relevant information without disturbance.

Make sure that you set production deployment rules for data sources and parameters defined in the semantic model.

## Update the production app

Deployment in a pipeline updates the workspace content, but it can also update the associated app through the [deployment pipelines API](#). It's not possible to update the app through the UI. You need to update the app manually. If you use an app for content distribution, don't forget to update the app after deploying to production so that end users are immediately able to use the latest version.

## Deploying into production using Git branches

As the repo serves as the 'single-source-of-truth', some teams might want to deploy updates into different stages directly from Git. This is possible with Git integration, with a few considerations:

- We recommend using release branches. You need to continuously change the connection of workspace to the new release branches before every deployment.
- If your build or release pipeline requires you to change the source code, or run scripts in a build environment before deployment to the workspace, then connecting the workspace to Git won't help you.

- After deploying to each stage, make sure to change all the configuration specific to that stage.

## Quick fixes to content

Sometimes there are issues in production that require a quick fix. Deploying a fix without testing it first is bad practice. Therefore, always implement the fix in the development stage and push it to the rest of the deployment pipeline stages. Deploying to the development stage allows you to check that the fix works before deploying it to production. Deploying across the pipeline takes only a few minutes.

If you're using deployment from Git, we recommend following the practices described in [Adopt a Git branching strategy](#).

## Related content

- [End to end lifecycle management tutorial](#)
- [Get started with Git integration](#)
- [Get started with deployment pipelines](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Automate your deployment pipeline by using APIs and Azure DevOps

Article • 11/02/2023

The Microsoft Fabric [deployment pipelines](#) tool enables business intelligence teams to build an efficient and reusable release process for their Fabric content.

To achieve continuous integration and continuous delivery (CI/CD) of content, many organizations use automation tools, including [Azure DevOps](#). Organizations that use Azure DevOps, can use the [Power BI automation tools](#) extension, which supports many of the deployment pipelines API operations.

You can use the [deployment pipelines Power BI REST APIs](#) to integrate Fabric into your organization's automation process. Here are a few examples of what can be done by using the APIs:

- Manage pipelines from start to finish, including creating a pipeline, assigning a workspace to any stage, and deploying and deleting the pipeline.
- Assign and unassign users to and from a pipeline.
- Integrate Fabric into familiar DevOps tools such as [Azure DevOps](#) or [GitHub Actions](#).
- Schedule pipeline deployments to happen automatically at a specific time.
- Deploy multiple pipelines at the same time.
- Cascade depending on pipeline deployments. If you have content that's connected across pipelines, you can make sure some pipelines are deployed before others.

## Deployment pipelines API functions

### Note

The deployment pipelines APIs currently only work for Power BI items.

The [deployment pipelines Power BI REST APIs](#) allow you to perform the following functions:

- **Get pipeline information** - Retrieve information about your pipelines and their content. Getting the pipeline information enables you to dynamically build the deployment API calls. You can also check the [status of a deployment](#) or the [deployment history](#).
- **Deploy** - The REST calls enable developers to use any type of deployment available in the Fabric service.
- **Create and delete pipelines** - Use [Create pipeline](#) and [Delete pipeline](#) to perform these operations.
- **Manage workspaces** - With [Assign workspace](#) and [Unassign workspace](#), you can assign and unassign workspaces to specific pipeline stages.
- **Manage pipeline users** - [Delete pipeline user](#) lets you remove a user from a pipeline. [Update pipeline user](#) allows you to add a user to your pipeline.

## Which deployments are supported by the APIs?

The APIs support the following deployment types:

- **Deploy all** - A single API call that deploys all the content in the workspace to the next stage in the pipeline. For this operation, use the [Deploy all](#) API.
- **Selective deploy** - Deploys only specific items, such as reports or dashboards, in the pipeline. For this operation, use the [Selective deploy](#) API.
- **Backward deploy** - Deploys new items to the previous stage. Backward deployment only works if the items that are deployed don't already exist in the target stage. For this operation, use either the [Deploy all](#) or the [Selective deploy](#) APIs, with `isBackwardDeployment` set to `True`.
- **Update App** - As part of the deployment API call, you can update the content of the app that's related to that stage. Updated items are automatically available to your end users, after a deployment has completed. For this operation, use either the [Deploy all](#) or the [Selective deploy](#) APIs, with [PipelineUpdateAppSettings](#).

## Before you begin

Before you use the deployment pipelines APIs, make sure you have the following:

- The [service principal](#), or the [user](#) that calls the APIs needs [pipeline and workspace permissions](#) and access to an [Microsoft Entra application](#).

- If you're going to use PowerShell scripts, install the Power BI PowerShell cmdlets [Install-Module MicrosoftPowerBIMgmt](#).

## Integrate your pipeline with Azure DevOps

To automate the deployment processes from within your [release pipeline in Azure DevOps](#), use one of these methods:

- **PowerShell** - The script signs into Fabric using a *service principal* or a *user*.
- **Power BI automation tools** - This extension works with a *service principal* or a *user*.

You can also use other [Power BI REST API](#) calls, to complete related operations such as importing a *.pbix* into the pipeline, updating data sources and parameters.

## Use the Power BI automation tools extension

The Power BI automation tools extension is an [open source](#)  Azure DevOps extension that provides a range of deployment pipelines operations that can be performed in Azure DevOps. The extension eliminates the need for APIs or scripts to manage pipelines. Each operation can be used individually to perform a task, such as creating a pipeline. Operations can be used together in an Azure DevOps pipeline to create a more complex scenario, such as creating a pipeline, assigning a workspace to the pipeline, adding users, and deploying.

After you add the [Power BI automation tools](#)  extension to DevOps, you need to create a service connection. The following connections are available:

- **Service principal** (recommended) - This connection authenticates by using a *service principal* and requires the Microsoft Entra app's secret and application ID. When you use this option, verify that the [service admin settings](#) for the service principal are enabled.
- **Username and password** – Configured as a generic service connection with a username and a password. This connection method doesn't support multi-factor authentication. We recommend that you use the service principal connection method because it doesn't require storing user credentials on Azure DevOps.

### Note

The Power BI automation tools extension uses an Azure DevOps service connection to store credentials. For more information, see [How we store your credentials for](#)

After you enable a service connection for your Azure DevOps Power BI automation tools, you can [create pipeline tasks](#). The extension includes the following deployment pipelines tasks:

- Create a new pipeline
- Assign a workspace to a pipeline stage
- Add a user to a deployment pipeline
- Add a user to a workspace
- Deploy content to a deployment pipeline
- Remove a workspace from a deployment pipeline
- Delete a pipeline

## Access the PowerShell samples

You can use the following PowerShell scripts to understand how to perform several automation processes. To view or copy the text in a PowerShell sample, use the links in this section.

You can also download the entire [PowerBI-Developer-Samples](#) GitHub folder.

- [Deploy all](#)
- [Selective deployment](#)
- [Wait for deployment](#)
- [End to end example of pipeline creation and backward deployment](#)
- [Assign an admin user to a pipeline](#)

## PowerShell example

This section describes an example PowerShell script that deploys a semantic model, report, and dashboard, from the development stage to the test stage. The script then checks whether the deployment was successful.

To run a PowerShell script that performs a deployment, you need the following components. You can add any of these parts into [tasks](#) in your Azure pipeline stages.

1. **Sign in** - Before you can deploy your content, you need to sign in to Fabric using a *service principal* or a *user*. Use the [Connect-PowerBIServiceAccount](#) command to sign in.
2. **Build your request body** - In this part of the script you specify which items (such as reports and dashboards) you're deploying.

PowerShell

```
$body = @{
 sourceStageOrder = 0 # The order of the source stage. Development (0), Test (1).
 datasets = @(
 @{sourceId = "Insert your dataset ID here" }
)
 reports = @(
 @{sourceId = "Insert your report ID here" }
)
 dashboards = @(
 @{sourceId = "Insert your dashboard ID here" }
)

 options = @{
 # Allows creating new item if needed on the Test stage workspace
 allowCreateArtifact = $TRUE

 # Allows overwriting existing item if needed on the Test stage workspace
 allowOverwriteArtifact = $TRUE
 }
} | ConvertTo-Json
```

3. **Deploy** - Here you perform the deployment.

PowerShell

```
$url = "pipelines/{0}/Deploy" -f "Insert your pipeline ID here"
$deployResult = Invoke-PowerBIRestMethod -Url $url -Method Post -Body
$body | ConvertFrom-Json
```

4. (Optional) **Deployment completion notification** - Because the deployment API is asynchronous, you can program the script to notify you when the deployment is complete.

## PowerShell

```
$url = "pipelines/{0}/Operations/{1}" -f "Insert you pipeline ID here",$deployResult.id
$operation = Invoke-PowerBIRestMethod -Url $url -Method Get |
ConvertFrom-Json
while($operation.Status -eq "NotStarted" -or $operation.Status -eq "Executing")
{
 # Sleep for 5 seconds
 Start-Sleep -s 5
 $operation = Invoke-PowerBIRestMethod -Url $url -Method Get |
ConvertFrom-Json
}
```

## Considerations and limitations

- Deployment by using APIs is subject to the same [limitations](#) as the deployment pipelines user interface.
- A *service principal* can't configure *OAuth* credentials. After you deploy new items, the signed in *service principal* becomes the owner of any deployed paginated reports and semantic models. In such cases, a refresh can't be completed.
- Deploying dataflows by using a *service principal* isn't supported.
- The maximum number of items that can be deployed in a single deployment is 300.
- The deployment pipelines APIs currently only support Power BI items.
- Creating a customized pipeline of 2-10 stages is currently supported only through the UI.

## Related content

- [Get started with deployment pipelines](#)
- [Deployment pipelines best practices](#)
- [Troubleshooting deployment pipelines](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Continuous integration and delivery for an Azure Synapse Analytics workspace

Article • 12/02/2022

Continuous integration (CI) is the process of automating the build and testing of code every time a team member commits a change to version control. Continuous delivery (CD) is the process of building, testing, configuring, and deploying from multiple testing or staging environments to a production environment.

In an Azure Synapse Analytics workspace, CI/CD moves all entities from one environment (development, test, production) to another environment. Promoting your workspace to another workspace is a two-part process. First, use an [Azure Resource Manager template \(ARM template\)](#) to create or update workspace resources (pools and workspace). Then, migrate artifacts like SQL scripts and notebooks, Spark job definitions, pipelines, datasets, and other artifacts by using [Synapse Workspace Deployment](#) tools in Azure DevOps or on GitHub.

This article outlines how to use an Azure DevOps release pipeline and GitHub Actions to automate the deployment of an Azure Synapse workspace to multiple environments.

## Prerequisites

To automate the deployment of an Azure Synapse workspace to multiple environments, the following prerequisites and configurations must be in place.

## Azure DevOps

- Prepare an Azure DevOps project for running the release pipeline.
- [Grant any users who will check in code Basic access at the organization level](#), so they can see the repository.
- Grant Owner permission to the Azure Synapse repository.
- Make sure that you've created a self-hosted Azure DevOps VM agent or use an Azure DevOps hosted agent.
- Grant permissions to [create an Azure Resource Manager service connection for the resource group](#).
- An Azure Active Directory (Azure AD) administrator must [install the Azure DevOps Synapse Workspace Deployment Agent extension in the Azure DevOps organization](#).

- Create or nominate an existing service account for the pipeline to run as. You can use a personal access token instead of a service account, but your pipelines won't work after the user account is deleted.

## GitHub

- Create a GitHub repository that contains the Azure Synapse workspace artifacts and the workspace template.
- Make sure that you've created a self-hosted runner or use a GitHub-hosted runner.

## Azure Active Directory

- If you're using a service principal, in Azure AD, create a service principal to use for deployment.
- If you're using a managed identity, enable the system-assigned managed identity on your VM in Azure as the agent or runner, and then add it to Azure Synapse Studio as Synapse admin.
- Use the Azure AD admin role to complete these actions.

## Azure Synapse Analytics

### Note

You can automate and deploy these prerequisites by using the same pipeline, an ARM template, or the Azure CLI, but these processes aren't described in this article.

- The "source" workspace that's used for development must be configured with a Git repository in Azure Synapse Studio. For more information, see [Source control in Azure Synapse Studio](#).
- Set up a blank workspace to deploy to:
  1. Create a new Azure Synapse workspace.
  2. Grant the VM agent and the service principal Contributor permission to the resource group in which the new workspace is hosted.
  3. In the workspace, don't configure the Git repository connection.
  4. In the Azure portal, find the new Azure Synapse workspace, and then grant Owner permission to yourself and to the user that will run the Azure DevOps pipeline Azure Synapse workspace.

5. Add the Azure DevOps VM agent and the service principal to the Contributor role for the workspace. (The role should have been inherited, but verify that it is.)
6. In the Azure Synapse workspace, go to **Studio > Manage > Access Control**. Add the Azure DevOps VM agent and the service principal to the workspace admin group.
7. Open the storage account that's used for the workspace. On the **Identity and access management** pane, add the VM agent and the service principal to the Storage Blob Data Contributor role.
8. Create a key vault in the support subscription, and ensure that both the existing workspace and the new workspace have at least GET and LIST permissions to the vault.
9. For the automated deployment to work, ensure that any connection strings that are specified in your linked services are in the key vault.

## Other prerequisites

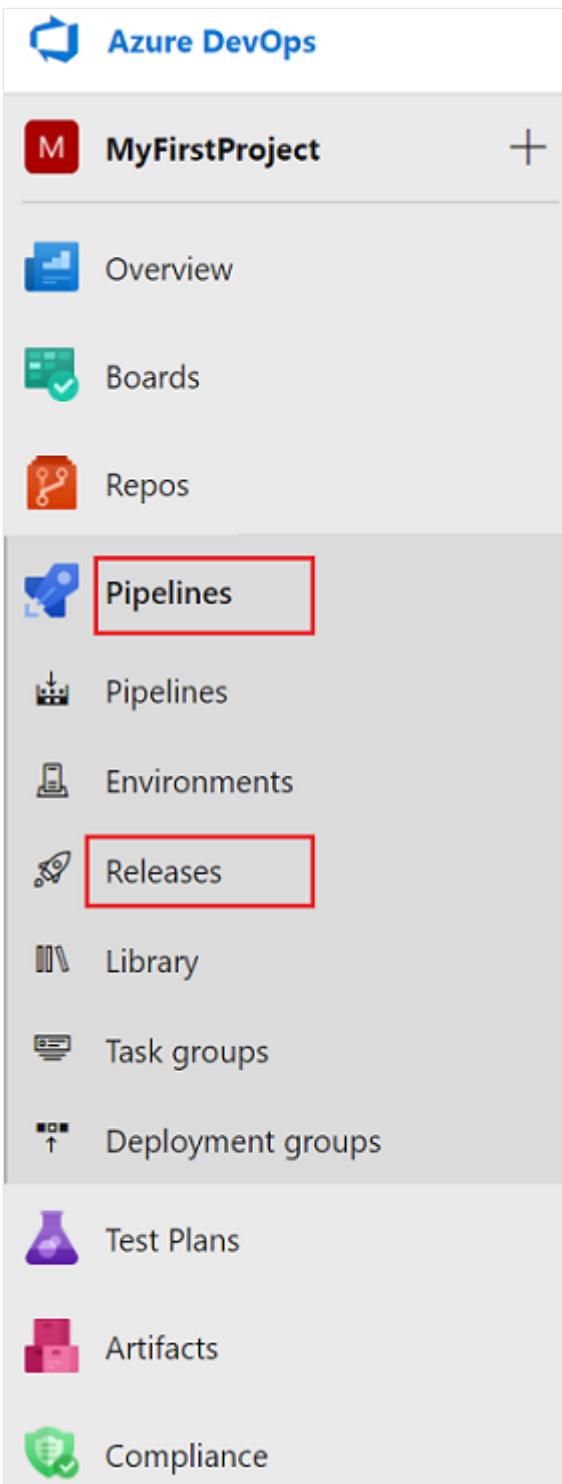
- Spark pools and self-hosted integration runtimes aren't created in a workspace deployment task. If you have a linked service that uses a self-hosted integration runtime, manually create the runtime in the new workspace.
- If the items in the development workspace are attached with the specific pools, make sure that you create or parameterize the same names for the pools in the target workspace in the parameter file.
- If your provisioned SQL pools are paused when you attempt to deploy, the deployment might fail.

For more information, see [CI/CD in Azure Synapse Analytics Part 4 - The release pipeline](#).

## Set up a release pipeline in Azure DevOps

In this section, you'll learn how to deploy an Azure Synapse workspace in Azure DevOps.

1. In [Azure DevOps](#), open the project you created for the release.
2. On the left menu, select **Pipelines > Releases**.



3. Select **New pipeline**. If you have existing pipelines, select **New > New release pipeline**.
4. Select the **Empty job template**.



Select a template

Or start with an [Empty job](#)

Search

Featured



Azure App Service deployment

Deploy your application to Azure App Service. Choose from Web App on Windows, Linux, containers, Function Apps, or WebJobs.

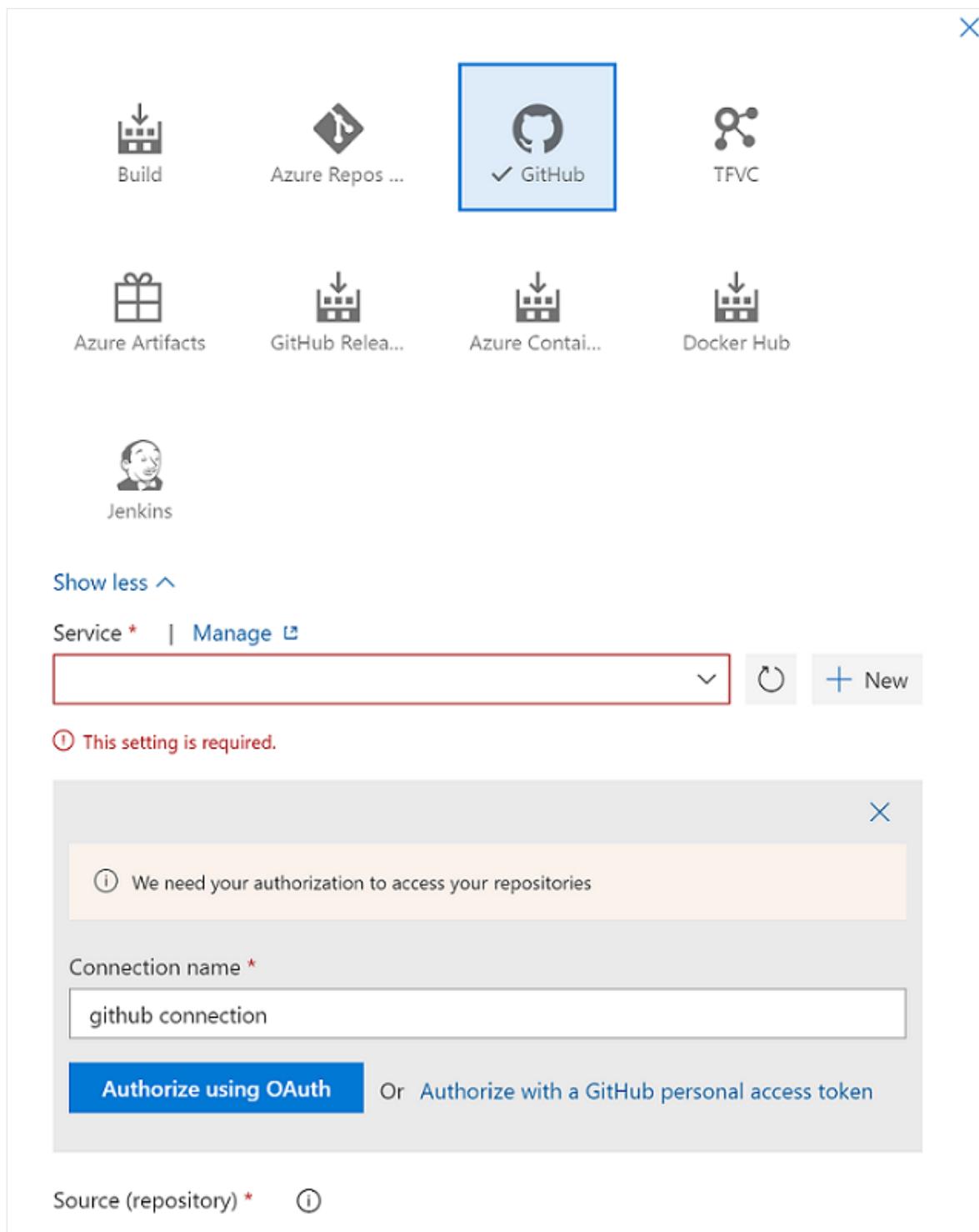


Deploy a Java app to Azure App Service

Deploy a Java application to an Azure Web App.

5. In **Stage name**, enter the name of your environment.

6. Select **Add artifact**, and then select the Git repository that's configured with Azure Synapse Studio in your development environment. Select the Git repository in which you manage your pools and workspace ARM template. If you use GitHub as the source, create a service connection for your GitHub account and pull repositories. For more information, see [service connections](#).



The screenshot shows the GitHub integration options in the Azure DevOps Services interface. The GitHub icon is highlighted with a blue border. Other options shown include Build, Azure Repos, TFVC, Azure Artifacts, GitHub Releases, Azure Container Registry, Docker Hub, and Jenkins. Below these, a 'Service' dropdown menu is open, showing 'Manage' and a 'New' button. A red box highlights the 'Authorize using OAuth' button, which is labeled as required. The 'Source (repository)' field is also visible.

Build      Azure Repos ...      GitHub      TFVC

Azure Artifacts      GitHub Relea...      Azure Contai...      Docker Hub

Jenkins

Show less ^

Service \* | Manage

① This setting is required.

① We need your authorization to access your repositories

Connection name \*

github connection

Authorize using OAuth      Or      Authorize with a GitHub personal access token

Source (repository) \*

7. Select the resource ARM template branch. For the **Default version**, select **Latest from default branch**.

All pipelines > **New release pipeline**

Pipeline Tasks Variables Retention Options History

Source type

Build **Azure Re...** GitHub TFVC

Artifacts | + Add

Artifacts\_source

Stages | + Add

Stage 1 1 job, 0 task

5 more artifact types

Project \* MyFirstProject

Source (repository) \* MyFirstProject

Default branch \* ResourceArmTemplate

Default version \* Latest from the default branch

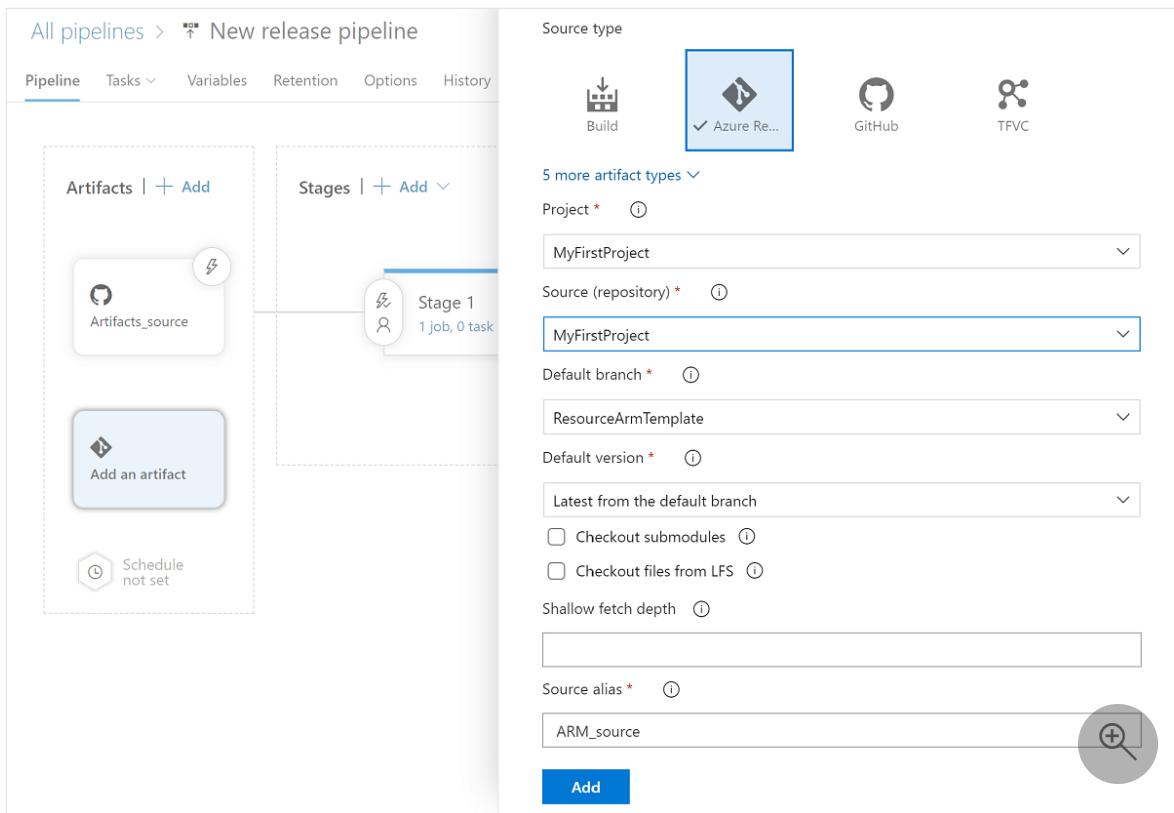
Checkout submodules

Checkout files from LFS

Shallow fetch depth

Source alias \* ARM\_source

Add



8. For the artifacts **Default branch**, select the repository **publish branch** or other non-publish branches which include Synapse artifacts. By default, the publish branch is **workspace\_publish**. For the **Default version**, select **Latest from default branch**.

Source type

Build      Azure Repos ...      **GitHub**      TFVC

5 more artifact types ▾

Service \* | Manage 

github connection   

Source (repository) \* 

Devworkspacegit 

Default branch \* 

workspace\_publish 

Default version \* 

Latest from the default branch 

Checkout submodules 

Checkout files from LFS 

Shallow fetch depth 

Source alias \* 

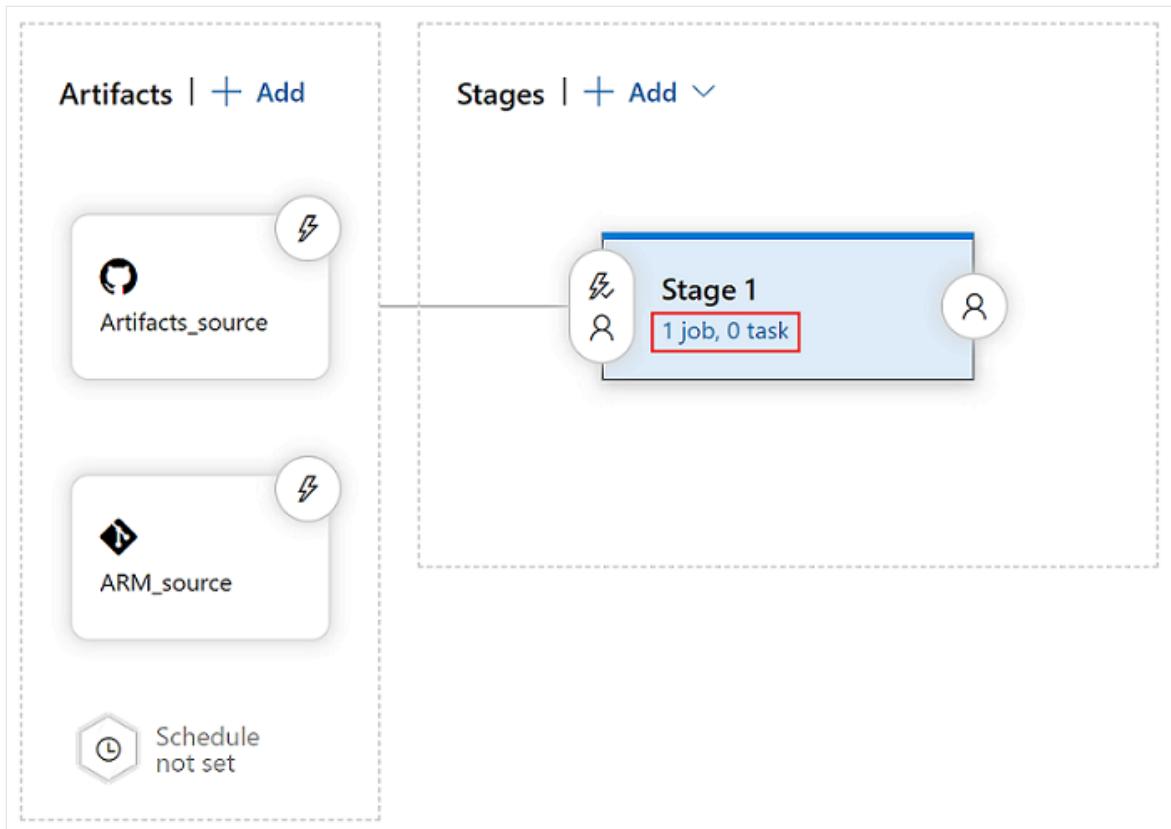
Artifacts\_source

**Add**

## Set up a stage task for an ARM template to create and update a resource

If you have an ARM template that deploys a resource, such as an Azure Synapse workspace, a Spark and SQL pool, or a key vault, add an Azure Resource Manager deployment task to create or update those resources:

1. In the stage view, select **View stage tasks**.



2. Create a new task. Search for **ARM Template Deployment**, and then select **Add**.
3. On the deployment **Tasks** tab, select the subscription, resource group, and location for the workspace. Provide credentials if necessary.
4. For **Action**, select **Create or update resource group**.
5. For **Template**, select the ellipsis button (...). Go to the ARM template of the workspace.
6. For **Template parameters**, select ... to choose the parameters file.

All pipelines >  New release pipeline

Save  Create release  View releases ...

Pipeline Tasks  Variables  Retention  Options 

Test-env-ARMResource 

Agent job  

 ARM Template deployment: Resource Group sc... 

 Azure PowerShell script: InlineScript 

Resource Group 

Azure Resource Manager connection \*  

Azure Subscription Service connection  

Subscription \* 

Subscription  

Action \* 

Create or update resource group 

Resource group \* 

test-env  

Location \* 

West US  

Template 

Template location \* 

Linked artifact  

Template \* 

`$(System.DefaultWorkingDirectory)/Resources-source/ARM resource/template.json`  

Template parameters 

`$(System.DefaultWorkingDirectory)/Resources-source/ARM resource/parameters.json`  

7. For **Override template parameters**, select ..., and then enter the parameter values you want to use for the workspace.

8. For **Deployment mode**, select **Incremental**.

9. (Optional) Add **Azure PowerShell** for the grant and update the workspace role assignment. If you use a release pipeline to create an Azure Synapse workspace, the pipeline's service principal is added as the default workspace admin. You can run PowerShell to grant other accounts access to the workspace.

All pipelines >  New release pipeline

Save  Create release  View releases ...

Pipeline Tasks  Variables  Retention  Options 

Test-env-ARMResource 

Agent job  

 ARM Template deployment: Resource Group sc... 

 Azure PowerShell script: Role assignment 

Display name \* 

Azure PowerShell script: Role assignment 

Azure Subscription \*  

Target workspace subscription  

Scoped to subscription \* 

Script Type 

Script File Path  Inline Script

Inline Script 

```
You can write your azure powershell scripts inline here.
You can also pass predefined and custom variables to this script using arguments
Install-Module Az_Synapse -RequiredVersion 0.2.0 -Scope CurrentUser -Force
New-AzSynapseRoleAssignment -WorkspaceName targetworkspace -RoleDefinitionName "Workspace Admin" -ObjectId "00000000-0000-0000-000000000000"
```

ErrorActionPreference 

Stop  

Fail on Standard Error 

Azure PowerShell version options 

Azure PowerShell Version 

Latest installed version  Specify other version

Preferred Azure PowerShell Version \* 

3.1.0  

## ⚠️ Warning

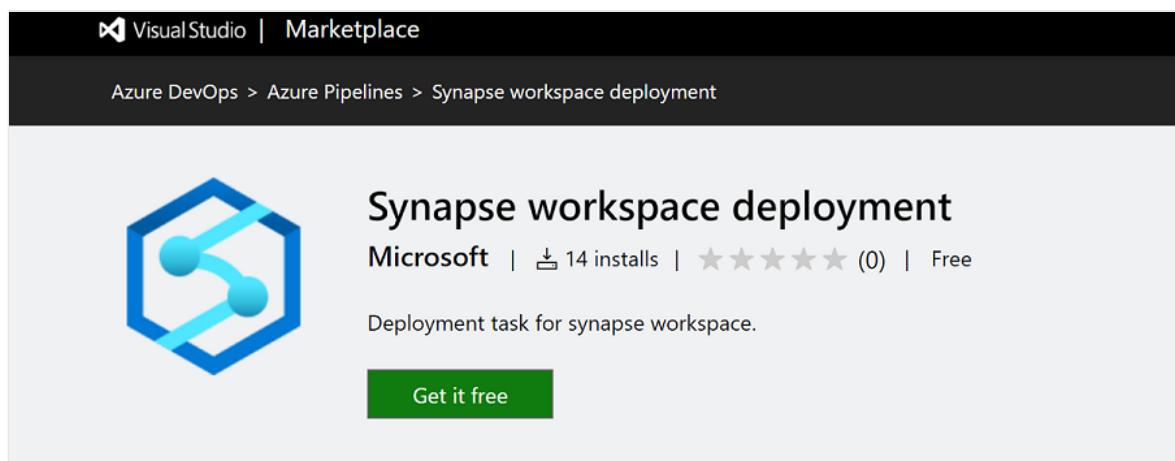
In complete deployment mode, resources in the resource group that aren't specified in the new ARM template are *deleted*. For more information, see [Azure Resource Manager deployment modes](#).

## Set up a stage task for Azure Synapse artifacts deployment

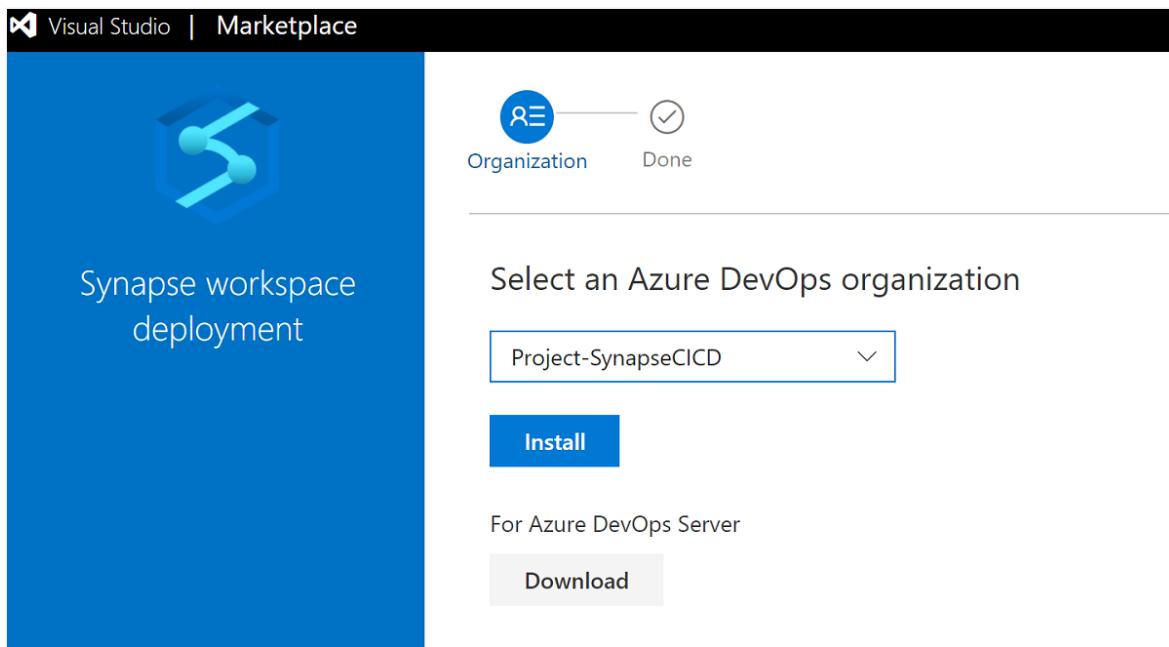
Use the [Synapse workspace deployment](#) extension to deploy other items in your Azure Synapse workspace. Items that you can deploy include datasets, SQL scripts and notebooks, spark job definitions, integration runtime, data flow, credentials, and other artifacts in workspace.

### Install and add deployment extension

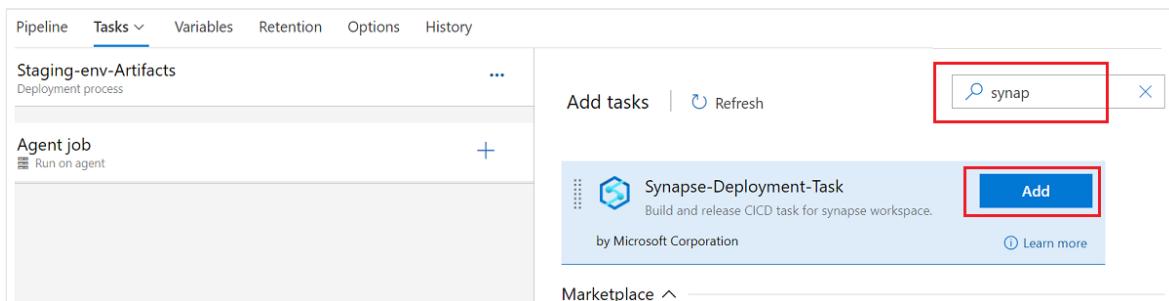
1. Search for and get the extension from [Visual Studio Marketplace](#).



2. Select the Azure DevOps organization in which you want to install the extension.



3. Make sure that the Azure DevOps pipeline's service principal has been granted the Subscription permission and is assigned as the Synapse workspace admin for the workspace.
4. To create a new task, search for **Synapse workspace deployment**, and then select **Add**.



## Configure the deployment task

The deployment task supports 3 types of operations, validate only, deploy and validate and deploy.

### ⓘ Note

This workspace deployment extension is not backward compatible. Please make sure that the latest version is installed and used. You can read the release note in [overview](#) in Azure DevOps and the [latest version](#) in GitHub action.

**Validate** is to validate the Synapse artifacts in non-publish branch with the task and generate the workspace template and parameter template file. The validation operation

only works in the YAML pipeline. The sample YAML file is as below:

```
YAML

pool:
 vmImage: ubuntu-latest

resources:
 repositories:
 - repository: <repository name>
 type: git
 name: <name>
 ref: <user/collaboration branch>

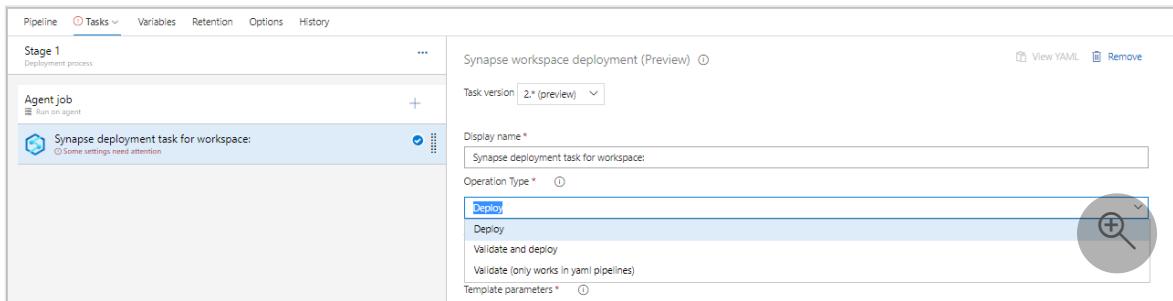
steps:
 - checkout: <name>
 - task: Synapse workspace deployment@2
 continueOnError: true
 inputs:
 operation: 'validate'
 ArtifactsFolder: '$(System.DefaultWorkingDirectory)/ArtifactFolder'
 TargetWorkspaceName: '<target workspace name>'
```

**Deploy** The inputs of the operation deploy include Synapse workspace template and parameter template, which can be created after publishing in the workspace publish branch or after the validation. It is same as the version 1.x.

**Validate and deploy** can be used to directly deploy the workspace from non-publish branch with the artifact root folder.

You can choose the operation types based on the use case. Following part is an example of the deploy.

1. In the task, select the operation type as **Deploy**.

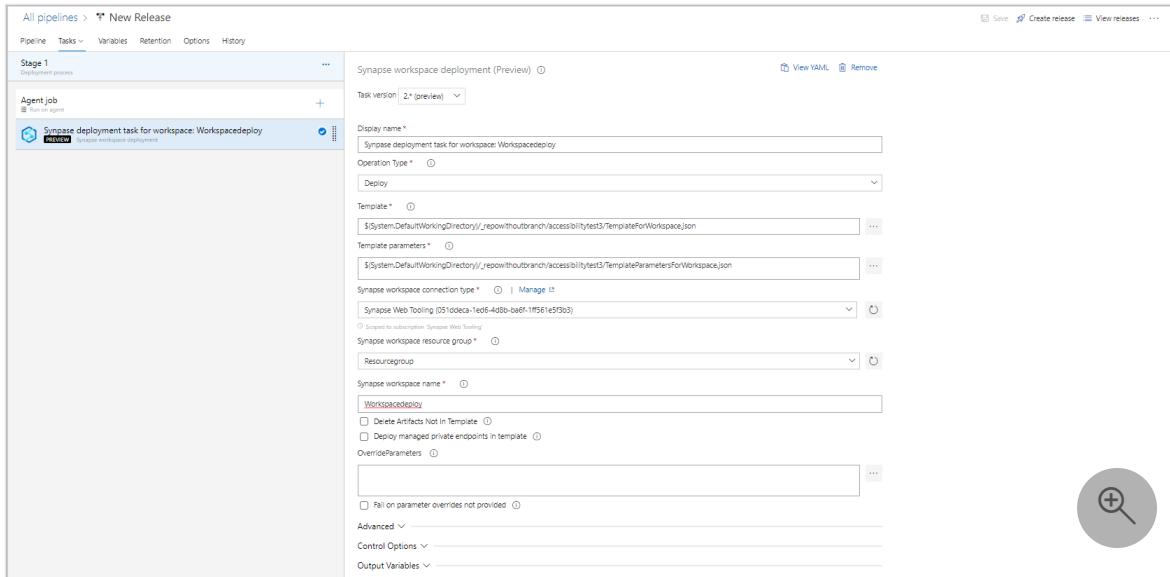


2. In the task, next to **Template**, select ... to choose the template file.

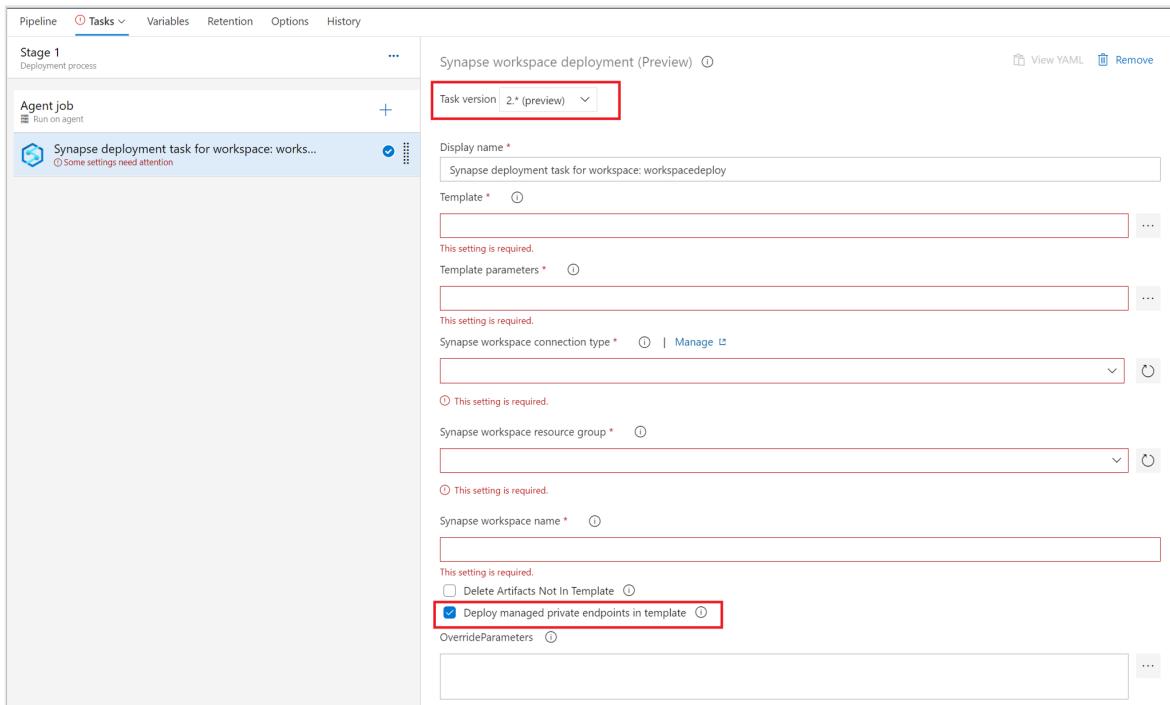
3. Next to **Template parameters**, select ... to choose the parameters file.

4. Select a connection, resource group, and name for the workspace.

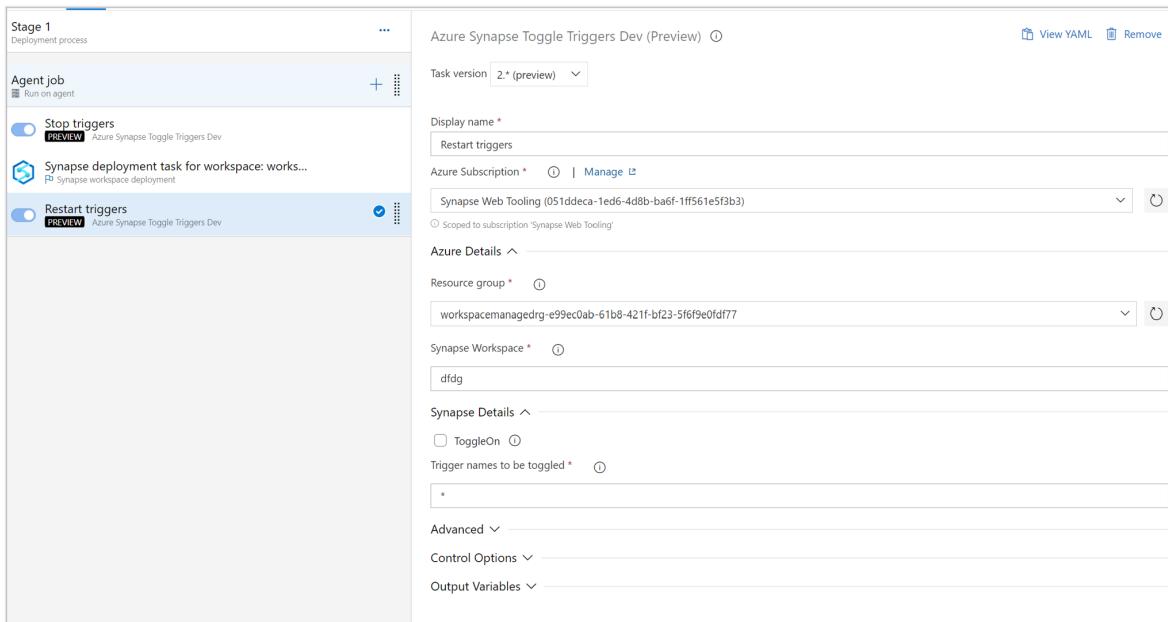
5. Next to **Override template parameters**, select ... . Enter the parameter values you want to use for the workspace, including connection strings and account keys that are used in your linked services. For more information, see [CI/CD in Azure Synapse Analytics](#).



6. The deployment of managed private endpoint is only supported in version 2.x. please make sure you select the right version and check the **Deploy managed private endpoints in template**.



7. To manage triggers, you can use trigger toggle to stop the triggers before deployment. And you can also add a task to restart the triggers after the deployment task.

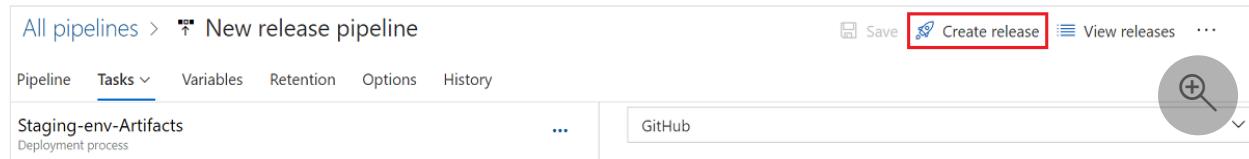


## ⓘ Important

In CI/CD scenarios, the integration runtime type in different environments must be the same. For example, if you have a self-hosted integration runtime in the development environment, the same integration runtime must be self-hosted in other environments, such as in test and production. Similarly, if you're sharing integration runtimes across multiple stages, the integration runtimes must be linked and self-hosted in all environments, such as in development, test, and production.

## Create a release for deployment

After you save all changes, you can select **Create release** to manually create a release. To learn how to automate release creation, see [Azure DevOps release triggers](#).



## Set up a release in GitHub Actions

In this section, you'll learn how to create GitHub workflows by using GitHub Actions for Azure Synapse workspace deployment.

You can use the [GitHub Actions for Azure Resource Manager template](#) to automate deploying an ARM template to Azure for the workspace and compute pools.

# Workflow file

Define a GitHub Actions workflow in a YAML (.yml) file in the `./github/workflows/` path in your repository. The definition contains the various steps and parameters that make up the workflow.

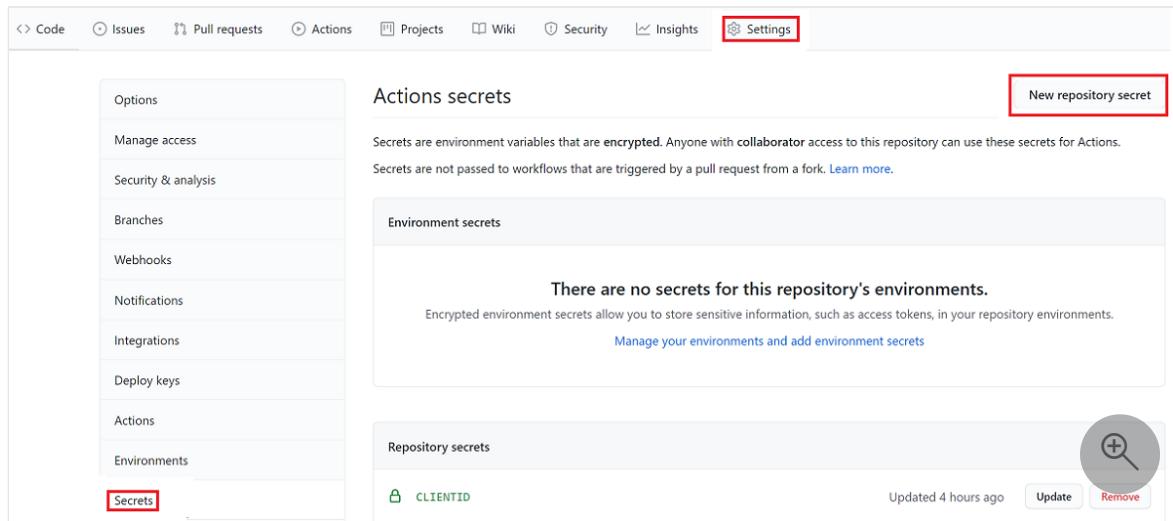
The .yml file has two sections:

Section	Tasks
Authentication	1. Define a service principal. 2. Create a GitHub secret.
Deploy	Deploy the workspace artifacts.

## Configure GitHub Actions secrets

GitHub Actions secrets are environment variables that are encrypted. Anyone who has Collaborator permission to this repository can use these secrets to interact with Actions in the repository.

1. In the GitHub repository, select the **Settings** tab, and then select **Secrets > New repository secret**.



2. Add a new secret for the client ID, and add a new client secret if you use the service principal for deployment. You can also choose to save the subscription ID and tenant ID as secrets.

## Add your workflow

In your GitHub repository, go to **Actions**.

1. Select **Set up your workflow yourself**.
2. In the workflow file, delete everything after the `on:` section. For example, your remaining workflow might look like this example:

```

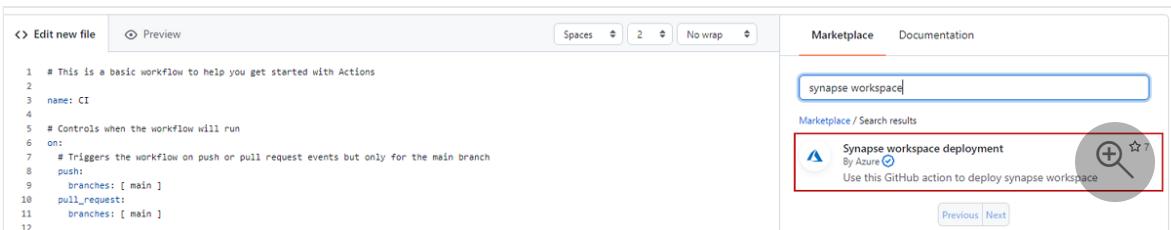
YAML

name: CI

on:
 push:
 branches: [master]
 pull_request:
 branches: [master]

```

3. Rename your workflow. On the **Marketplace** tab, search for the Synapse workspace deployment action, and then add the action.



4. Set the required values and the workspace template:

```

YAML

name: workspace deployment

on:
 push:
 branches: [publish_branch]
jobs:
 release:
 # You also can use the self-hosted runners.
 runs-on: windows-latest
 steps:
 # Checks out your repository under $GITHUB_WORKSPACE, so your
 # job can access it.
 - uses: actions/checkout@v2
 - uses: azure/synapse-workspace-deployment@release-1.0
 with:
 TargetWorkspaceName: 'target workspace name'
 TemplateFile: './path of the TemplateForWorkspace.json'
 ParametersFile: './path of the
TemplateParametersForWorkspace.json'
 OverrideArmParameters: './path of the parameters.yaml'
 environment: 'Azure Public'
 resourceGroup: 'target workspace resource group'

```

```

clientId: ${secrets.CLIENTID}
clientSecret: ${secrets.CLIENTSECRET}
subscriptionId: 'subscriptionId of the target workspace'
tenantId: 'tenantId'
DeleteArtifactsNotInTemplate: 'true'
managedIdentity: 'False'

```

5. You're ready to commit your changes. Select **Start commit**, enter the title, and then add a description (optional). Then, select **Commit new file**.



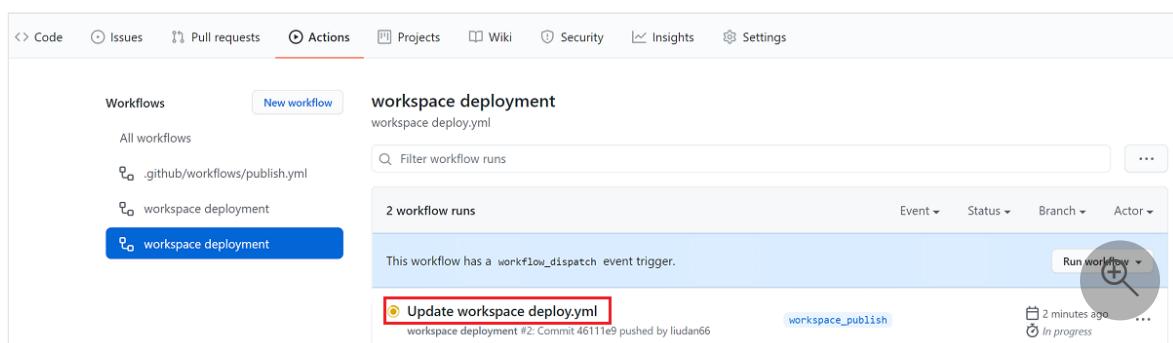
The file appears in the `.github/workflows` folder in your repository.

### ⓘ Note

Managed identity is supported only with self-hosted VMs in Azure. Be sure to set the runner to self-hosted. Enable the system-assigned managed identity for your VM and add it to Azure Synapse Studio as Synapse admin.

## Review your deployment

1. In your GitHub repository, go to **Actions**.
2. To see detailed logs of your workflow's run, open the first result:



# Create custom parameters in the workspace template

If you use automated CI/CD and want to change some properties during deployment, but the properties aren't parameterized by default, you can override the default parameter template.

To override the default parameter template, create a custom parameter template named `template-parameters-definition.json` in the root folder of your Git branch. You must use this exact file name. When Azure Synapse workspace publishes from the collaboration branch or the deployment task validates the artifacts in other branches, it reads this file and uses its configuration to generate the parameters. If Azure Synapse workspace doesn't find that file, it uses the default parameter template.

## Custom parameter syntax

You can use the following guidelines to create a custom parameters file:

- Enter the property path under the relevant entity type.
- Setting a property name to `*` indicates that you want to parameterize all properties under the property (only down to the first level, not recursively). You can set exceptions to this configuration.
- Setting the value of a property as a string indicates that you want to parameterize the property. Use the format `<action>:<name>:<stype>`.
  - `<action>` can be one of these characters:
    - `=` means keep the current value as the default value for the parameter.
    - `-` means don't keep the default value for the parameter.
    - `|` is a special case for secrets from Azure Key Vault for connection strings or keys.
  - `<name>` is the name of the parameter. If it's blank, it takes the name of the property. If the value starts with a `-` character, the name is shortened. For example, `AzureStorage1_properties_typeProperties_connectionString` would be shortened to `AzureStorage1_connectionString`.
  - `<stype>` is the type of parameter. If `<stype>` is blank, the default type is `string`. Supported values: `string`, `securestring`, `int`, `bool`, `object`, `secureobject` and `array`.
- Specifying an array in the file indicates that the matching property in the template is an array. Azure Synapse iterates through all the objects in the array by using the definition that's specified. The second object, a string, becomes the name of the property, which is used as the name for the parameter for each iteration.

- A definition can't be specific to a resource instance. Any definition applies to all resources of that type.
- By default, all secure strings (such as Key Vault secrets) and secure strings (such as connection strings, keys, and tokens) are parameterized.

## Parameter template definition example

Here's an example of what a parameter template definition looks like:

JSON

```
{
 "Microsoft.Synapse/workspaces/notebooks": {
 "properties": {
 "bigDataPool": {
 "referenceName": "="
 }
 }
 },
 "Microsoft.Synapse/workspaces/sqlscripts": {
 "properties": {
 "content": {
 "currentConnection": {
 "*": "_"
 }
 }
 }
 },
 "Microsoft.Synapse/workspaces/pipelines": {
 "properties": {
 "activities": [
 {
 "typeProperties": {
 "waitTimeInSeconds": "-::int",
 "headers": "=::object",
 "activities": [
 {
 "typeProperties": {
 "url": "-:-webUrl:string"
 }
 }
]
 }
 }
]
 }
 },
 "Microsoft.Synapse/workspaces/integrationRuntimes": {
 "properties": {
 "typeProperties": {
 "*": "="
 }
 }
 }
}
```

```
},
"Microsoft.Synapse/workspaces/triggers": {
 "properties": {
 "typeProperties": {
 "recurrence": {
 "*": "=",
 "interval": "=:triggerSuffix:int",
 "frequency": "=:-freq"
 },
 "maxConcurrency": "="
 }
 }
},
"Microsoft.Synapse/workspaces/linkedServices": {
 "*": {
 "properties": {
 "typeProperties": {
 "accountName": "=",
 "username": "=",
 "connectionString": "|:-connectionString:secureString",
 "secretAccessKey": "|"
 }
 }
 }
},
"AzureDataLakeStore": {
 "properties": {
 "typeProperties": {
 "dataLakeStoreUri": "="
 }
 }
},
"AzureKeyVault": {
 "properties": {
 "typeProperties": {
 "baseUrl": "|:baseUrl:secureString"
 },
 "parameters": {
 "KeyVaultURL": {
 "type": "=",
 "defaultValue": "|:defaultValue:secureString"
 }
 }
 }
},
"Microsoft.Synapse/workspaces/datasets": {
 "*": {
 "properties": {
 "typeProperties": {
 "folderPath": "=",
 "fileName": "="
 }
 }
 }
},
}];
```

```
"Microsoft.Synapse/workspaces/credentials" : {
 "properties": {
 "typeProperties": {
 "resourceId": "="
 }
 }
}
```

Here's an explanation of how the preceding template is constructed, by resource type.

#### notebooks

- Any property in the `properties/bigDataPool/referenceName` path is parameterized with its default value. You can parameterize an attached Spark pool for each notebook file.

#### sqlscripts

- In the `properties/content/currentConnection` path, both the `poolName` and the `databaseName` properties are parameterized as strings without the default values in the template.

#### pipelines

- Any property in the `activities/typeProperties/waitTimeInSeconds` path is parameterized. Any activity in a pipeline that has a code-level property named `waitTimeInSeconds` (for example, the `Wait` activity) is parameterized as a number, with a default name. The property won't have a default value in the Resource Manager template. Instead, the property will be required input during Resource Manager deployment.
- The `headers` property (for example, in a `Web` activity) is parameterized with the `object` type (Object). The `headers` property has a default value that is the same value as the source factory.

#### integrationRuntimes

- All properties in the `typeProperties` path are parameterized with their respective default values. For example, two properties are under `IntegrationRuntimes` type properties: `computeProperties` and `ssisProperties`. Both property types are created with their respective default values and types (Object).

#### triggers

- Under `typeProperties`, two properties are parameterized:

- The `maxConcurrency` property has a default value and is the `string` type. The default parameter name of the `maxConcurrency` property is `<entityName>_properties_typeProperties_maxConcurrency`.
- The `recurrence` property also is parameterized. All properties under the `recurrence` property are set to be parameterized as strings, with default values and parameter names. An exception is the `interval` property, which is parameterized as the `int` type. The parameter name is suffixed with `<entityName>_properties_typeProperties_recurrence_triggerSuffix`. Similarly, the `freq` property is a string and is parameterized as a string. However, the `freq` property is parameterized without a default value. The name is shortened and suffixed, such as `<entityName>_freq`.

### `linkedServices`

- Linked services are unique. Because linked services and datasets have a wide range of types, you can provide type-specific customization. In the preceding example, for all linked services of the `AzureDataLakeStore` type, a specific template is applied. For all others (identified through the use of the `*` character), a different template is applied.
- The `connectionString` property is parameterized as a `securestring` value. It doesn't have a default value. The parameter name is shortened and suffixed with `connectionString`.
- The `secretAccessKey` property is parameterized as an `AzureKeyVaultSecret` value (for example, in an Amazon S3 linked service). The property is automatically parameterized as an Azure Key Vault secret and fetched from the configured key vault. You also can parameterize the key vault itself.

### `datasets`

- Although you can customize types in datasets, an explicit `*`-level configuration isn't required. In the preceding example, all dataset properties under `typeProperties` are parameterized.

## Best practices for CI/CD

If you're using Git integration with your Azure Synapse workspace and you have a CI/CD pipeline that moves your changes from development to test, and then to production, we recommend these best practices:

- **Integrate only the development workspace with Git.** If you use Git integration, integrate only your *development* Azure Synapse workspace with Git. Changes to

test and production workspaces are deployed via CI/CD and don't need Git integration.

- **Prepare pools before you migrate artifacts.** If you have a SQL script or notebook attached to pools in the development workspace, use the same name for pools in different environments.
- **Sync versioning in infrastructure as code scenarios.** To manage infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, use the same versioning that the DevOps team uses for source code.
- **Review Azure Data Factory best practices.** If you use Data Factory, see the [best practices for Data Factory artifacts](#).

## Troubleshoot artifacts deployment

### Use the Synapse workspace deployment task to deploy Synapse artifacts

In Azure Synapse, unlike in Data Factory, artifacts aren't Resource Manager resources. You can't use the ARM template deployment task to deploy Azure Synapse artifacts. Instead, use the Synapse workspace deployment task to deploy the artifacts, and use ARM deployment task for ARM resources (pools and workspace) deployment. Meanwhile this task only supports Synapse templates where resources have type Microsoft.Synapse. And with this task, users can deploy changes from any branches automatically without manual clicking the publish in Synapse studio. The following are some frequently raised issues.

#### 1. Publish failed: workspace arm file is more than 20MB

There is a file size limitation in git provider, for example, in Azure DevOps the maximum file size is 20Mb. Once the workspace template file size exceeds 20Mb, this error happens when you publish changes in Synapse studio, in which the workspace template file is generated and synced to git. To solve the issue, you can use the Synapse deployment task with **validate** or **validate and deploy** operation to save the workspace template file directly into the pipeline agent and without manual publish in synapse studio.

#### 2. Unexpected token error in release

If your parameter file has parameter values that aren't escaped, the release pipeline fails to parse the file and generates an `unexpected token` error. We suggest that you override parameters or use Key Vault to retrieve parameter values. You also can use double escape characters to resolve the issue.

### **3. Integration runtime deployment failed**

If you have the workspace template generated from a managed Vnet enabled workspace and try to deploy to a regular workspace or vice versa, this error happens.

### **4. Unexpected character encountered while parsing value**

The template can not be parsed the template file. Try by escaping the back slashes, eg.  
`\Test01\Test`

### **5. Failed to fetch workspace info, Not found**

The target workspace info is not correctly configured. Please make sure the service connection which you have created, is scoped to the resource group which has the workspace.

### **6. Artifact deletion failed**

The extension will compare the artifacts present in the publish branch with the template and based on the difference it will delete them. Please make sure you are not trying to delete any artifact which is present in publish branch and some other artifact has a reference or dependency on it.

### **8. Deployment failed with error: json position 0**

If you were trying to manually update the template, this error would happen. Please make sure that you have not manually edited the template.

### **9. The document creation or update failed because of invalid reference**

The artifact in synapse can be referenced by another one. If you have parameterized an attribute which is a referenced in an artifact, please make sure to provide correct and non null value to it

## 10. Failed to fetch the deployment status in notebook deployment

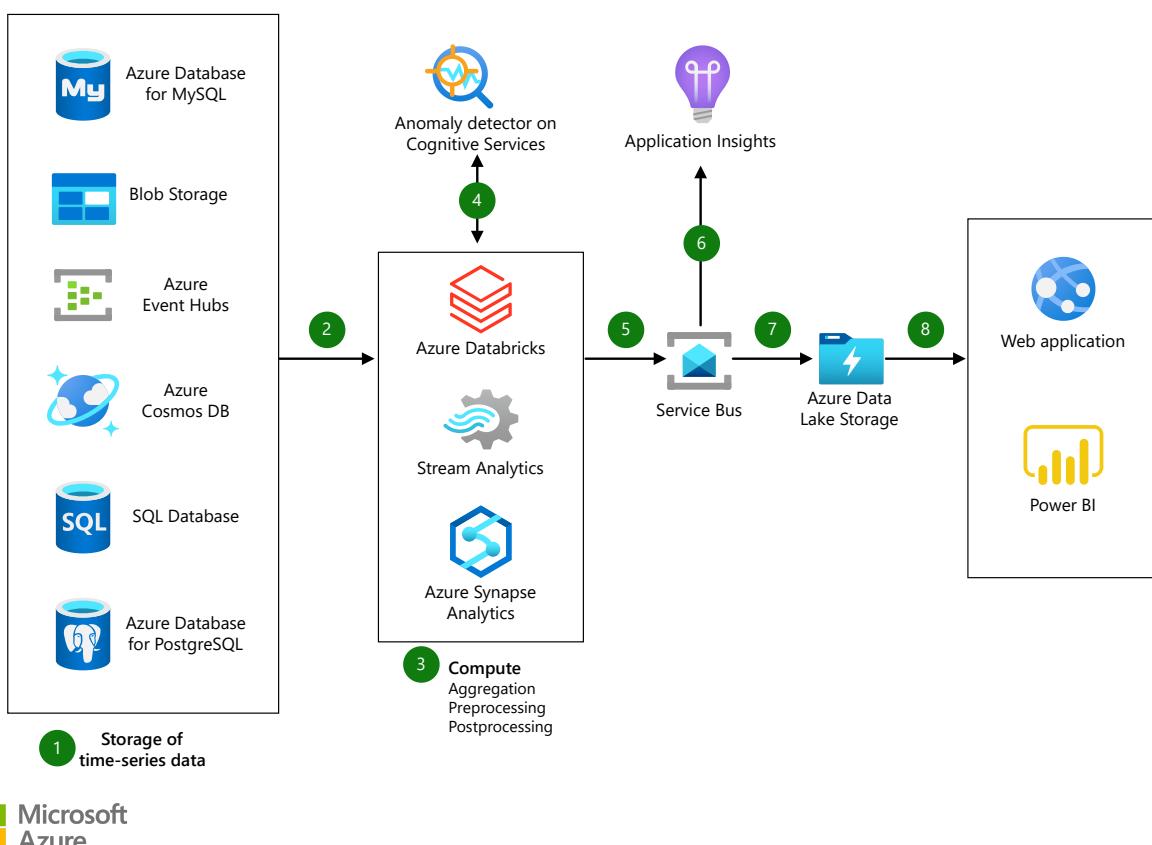
The notebook you are trying to deploy is attached to a spark pool in the workspace template file, while in the deployment the pool does not exist in the target workspace. If you don't parameterize the pool name, please make sure that having the same name for the pools between environments.

# Anomaly detector process

Azure Databricks   Azure Service Bus   Azure Storage Accounts

This article presents an architecture for a near real-time implementation of an anomaly detection process.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

1. Time-series data can come from multiple sources, such as Azure Database for MySQL, Blob storage, Event Hubs, Azure Cosmos DB, SQL Database, and Azure Database for PostgreSQL.
2. Data is ingested into compute from various storage sources to be monitored by Anomaly Detector.
3. Databricks helps aggregate, sample, and compute the raw data to generate the time with the detected results. Databricks is capable of processing stream and

static data. Stream analytics and Azure Synapse can be alternatives based on the requirements.

4. The anomaly detector API detects anomalies and returns the results to compute.
5. The anomaly-related metadata is queued.
6. Application Insights picks the message from the message queue based on the anomaly-related metadata and sends an alert about the anomaly.
7. The results are stored in Azure Data Lake Service Gen2.
8. Web applications and Power BI can visualize the results of the anomaly detection.

## Components

Key technologies used to implement this architecture:

- [Service Bus](#): Reliable cloud messaging as a service (MaaS) and simple hybrid integration.
- [Azure Databricks](#): Fast, easy, and collaborative Apache Spark-based analytics service.
- [Power BI](#): Interactive data visualization BI tools.
- [Storage Accounts](#): Durable, highly available, and massively scalable cloud storage.
- [Cognitive Services](#): Cloud-based services with REST APIs and client library SDKs available to help you build cognitive intelligence into your applications.
- [Logic Apps](#): Serverless platform for building enterprise workflows that integrate applications, data, and services. In this architecture, the logic apps are triggered by HTTP requests.
- [Azure Data Lake Storage Gen2](#): Azure Data Lake Storage Gen2 provides file system semantics, file-level security, and scale.
- [Application Insights](#): Application Insights is a feature of Azure Monitor that provides extensible application performance management (APM) and monitoring for live web apps.

## Alternatives

- [Event Hubs with Kafka](#): An alternative to running your own Kafka cluster. This Event Hubs feature provides an endpoint that is compatible with Kafka APIs.
- [Azure Synapse Analytics](#): An analytics service that brings together enterprise data warehousing and big data analytics.
- [Azure Machine Learning](#): Build, train, deploy, and manage custom machine learning / anomaly detection models in a cloud-based environment.

# Scenario details

The Azure Cognitive Services Anomaly Detector API enables you to monitor and detect abnormalities in your time series data without having to know machine learning. The algorithms of the API adapt by automatically identifying and applying the best-fitting models to your time series data, regardless of industry, scenario, or data volume. They determine boundaries for anomaly detection, expected values, and anomalous data points.

## Potential use cases

Some areas that anomaly detection helps monitor:

- Bank fraud (finance industry)
- Structural defects (manufacturing industry)
- Medical problems (healthcare industry)

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

## Scalability

Most of the components used in this example scenario are managed services that will automatically scale.

For general guidance on designing scalable solutions, see the [performance efficiency checklist](#) in the Azure Architecture Center.

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

[Managed identities for Azure resources](#) are used to provide access to other resources internal to your account and then assigned to your Azure Functions. Allow those identities to access only requisite resources to ensure that nothing extra is exposed to your functions (and potentially to your customers).

For general guidance on designing secure solutions, see the [Azure Security Documentation](#).

## Resiliency

All of the components in this scenario are managed, so at a regional level they're all resilient automatically.

For general guidance on designing resilient solutions, see [Designing resilient applications for Azure](#).

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

To explore the cost of running this scenario, see the pre-filled calculator with all of the services. To see how the pricing would change for your particular use case, change the appropriate variables to match your expected traffic / data volumes.

We've provided three sample cost profiles based on the amount of traffic (we assume all images are 100 kb in size):

- [Example calculator](#): this pricing example is a calculator with all services in this architecture, except Power BI and custom alerting solution.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Ashish Chauhan](#) | Senior Cloud Solution Architect

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [Anomaly Detector API Documentation](#)
- [Interactive demo](#)

- Detect and visualize anomalies in your data with the Anomaly Detector API - Demo on Jupyter Notebook ↗
- Identify anomalies by routing data via IoT Hub to a built-in ML model in Azure Stream Analytics
- Recipe: Predictive maintenance with the Cognitive Services for Big Data
- Service Bus Documentation
- Azure Databricks Documentation
- Power BI Documentation
- Storage Documentation

## Related resources

- Quality assurance
- Supply chain track and trace
- Introduction to predictive maintenance in manufacturing
- Predictive maintenance solution
- Predictive maintenance with the intelligent IoT Edge
- Stream processing with fully managed open-source data engines
- Connected factory hierarchy service
- Connected factory signal pipeline

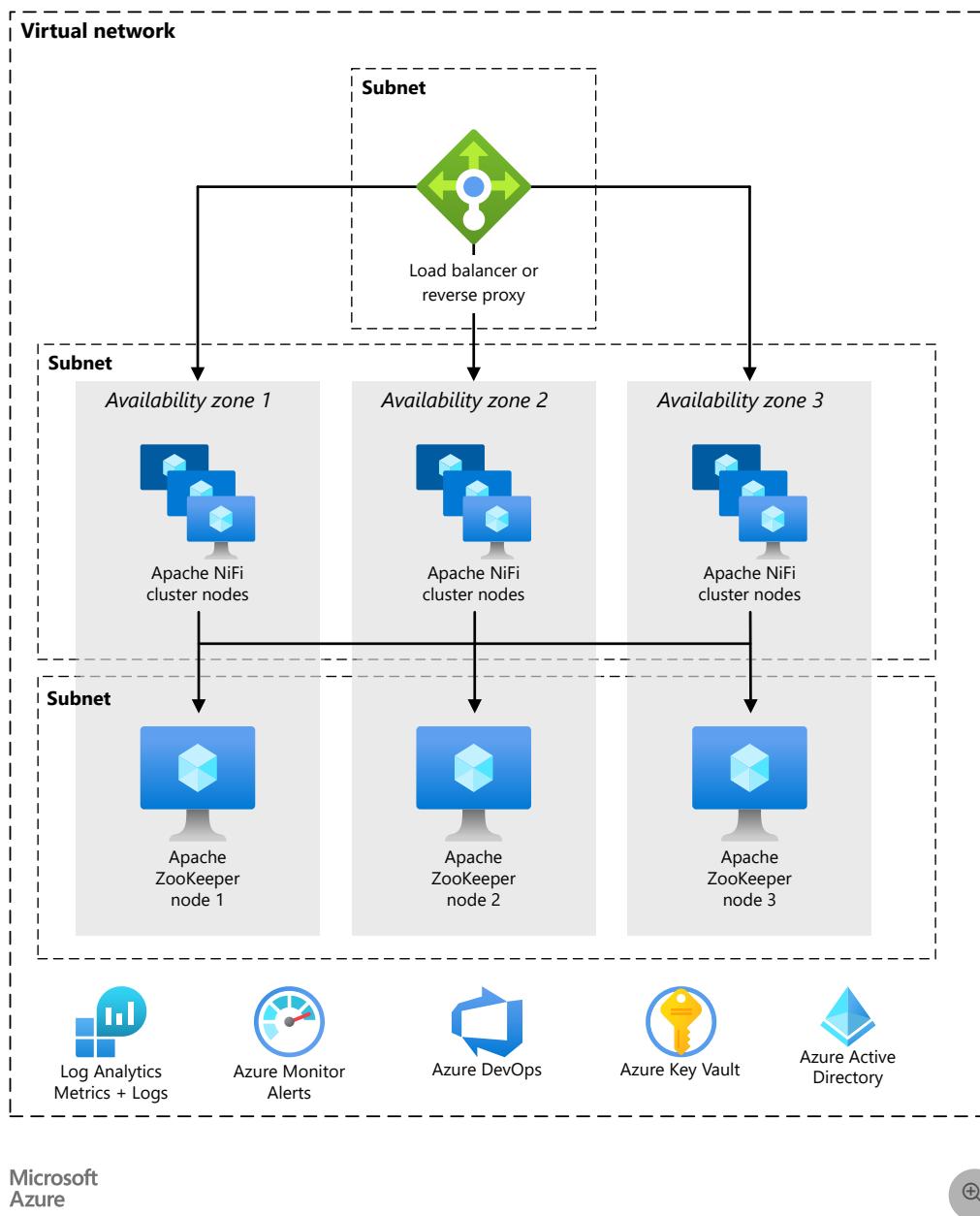
# Apache NiFi on Azure

Azure Application Gateway   Azure Log Analytics   Azure Virtual Machines   Azure Virtual Network   Azure Virtual Machine Scale Sets

This example scenario shows how to run [Apache NiFi](#) on Azure. NiFi provides a system for processing and distributing data.

Apache®, Apache NiFi®, and NiFi® are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

## Architecture



Download a [Visio file](#) of this architecture.

## Workflow

- The NiFi application runs on VMs in NiFi cluster nodes. The VMs are in a virtual machine scale set that the configuration deploys across availability zones.
- Apache ZooKeeper runs on VMs in a separate cluster. NiFi uses the ZooKeeper cluster for these purposes:
  - To choose a cluster coordinator node

- To coordinate the flow of data
- Azure Application Gateway provides layer-7 load balancing for the user interface that runs on the NiFi nodes.
- Monitor and its Log Analytics feature collect, analyze, and act on telemetry from the NiFi system. The telemetry includes NiFi system logs, system health metrics, and performance metrics.
- Azure Key Vault securely stores certificates and keys for the NiFi cluster.
- Microsoft Entra ID provides single sign-on and multifactor authentication.

## Components

- [NiFi](#) provides a system for processing and distributing data.
- [ZooKeeper](#) is an open-source server that manages distributed systems.
- [Virtual Machines](#) is an infrastructure-as-a-service (IaaS) offer. You can use Virtual Machines to deploy on-demand, scalable computing resources. Virtual Machines provides the flexibility of virtualization but eliminates the maintenance demands of physical hardware.
- [Azure Virtual Machine Scale Sets](#) provide a way to manage a group of load-balanced VMs. The number of VM instances in a set can automatically increase or decrease in response to demand or a defined schedule.
- [Availability zones](#) are unique physical locations within an Azure region. These high-availability offerings protect applications and data from datacenter failures.
- [Application Gateway](#) is a load balancer that manages traffic to web applications.
- [Monitor](#) collects and analyzes data on environments and Azure resources. This data includes app telemetry, such as performance metrics and activity logs. For more information, see [Monitoring considerations](#) later in this article.
- [Log Analytics](#) is an Azure portal tool that runs queries on Monitor log data. Log Analytics also provides features for charting and statistically analyzing query results.
- [Azure DevOps Services](#) provides services, tools, and environments for managing coding projects and deployments.
- [Key Vault](#) securely stores and controls access to a system's secrets, such as API keys, passwords, certificates, and cryptographic keys.
- [Microsoft Entra ID](#) is a cloud-based identity service that controls access to Azure and other cloud apps.

## Alternatives

- [Azure Data Factory](#) provides an alternative to this solution.
- Instead of Key Vault, you can use a comparable service to store system secrets.
- [Apache Airflow](#). See [how Airflow and NiFi are different](#).
- It is possible to use a supported enterprise NiFi alternative like [Cloudera Apache NiFi](#). The Cloudera offering is available through the [Azure Marketplace](#).

## Scenario details

In this scenario, NiFi runs in a clustered configuration across Azure Virtual Machines in a scale set. But most of this article's recommendations also apply to scenarios that run NiFi in single-instance mode on a single virtual machine (VM). The best practices in this article demonstrate a scalable, high-availability, and secure deployment.

## Potential use cases

NiFi works well for moving data and managing the flow of data:

- Connecting decoupled systems in the cloud
- Moving data in and out of Azure Storage and other data stores
- Integrating edge-to-cloud and hybrid-cloud applications with Azure IoT, Azure Stack, and Azure Kubernetes Service (AKS)

As a result, this solution applies to many areas:

- Modern data warehouses (MDWs) bring structured and unstructured data together at scale. They collect and store data from various sources, sinks, and formats. NiFi excels at ingesting data into Azure-based MDWs for the following reasons:
  - Over 200 processors are available for reading, writing, and manipulating data.
  - The system supports Storage services such as Azure Blob Storage, Azure Data Lake Storage, Azure Event Hubs, Azure Queue Storage, Azure Cosmos DB, and Azure Synapse Analytics.
  - Robust data provenance capabilities make it possible to implement compliant solutions. For information about capturing data provenance in the Log Analytics feature of Azure Monitor, see [Reporting considerations](#) later in this article.

- NiFi can run on a standalone basis on small-footprint devices. In such cases, NiFi makes it possible to process edge data and move that data to larger NiFi instances or clusters in the cloud. NiFi helps filter, transform, and prioritize edge data in motion, ensuring reliable and efficient data flows.
- Industrial IoT (IIoT) solutions manage the flow of data from the edge to the data center. That flow starts with data acquisition from industrial control systems and equipment. The data then moves to data management solutions and MDWs. NiFi offers capabilities that make it well suited for data acquisition and movement:
  - Edge data processing functionality
  - Support for protocols that IoT gateways and devices use
  - Integration with Event Hubs and Storage services

IoT applications in the areas of predictive maintenance and supply chain management can make use of this functionality.

## Recommendations

Keep the following points in mind when you use this solution:

### Recommended versions of NiFi

When you run this solution on Azure, we recommend using version 1.13.2+ of NiFi. You can run other versions, but they might require different configurations from the ones in this guide.

To install NiFi on Azure VMs, it's best to download the convenience binaries from the [NiFi downloads page](#). You can also build the binaries from [source code](#).

### Recommended versions of ZooKeeper

For this example workload, we recommend using versions 3.5.5 and later or 3.6.x of ZooKeeper.

You can install ZooKeeper on Azure VMs by using official convenience binaries or source code. Both are available on the [Apache ZooKeeper releases page](#).

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

For information on configuring NiFi, see the [Apache NiFi System Administrator's Guide](#). Also keep these considerations in mind when you implement this solution.

### Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

- Use the [Azure Pricing Calculator](#) to estimate the cost of the resources in this architecture.
- For an estimate that includes all the services in this architecture except the custom alerting solution, see this [sample cost profile](#).

### VM considerations

The following sections provide a detailed outline of how to configure the NiFi VMs:

#### VM size

This table lists recommended VM sizes to start with. For most general-purpose data flows, Standard\_D16s\_v3 is best. But each data flow in NiFi has different requirements. Test your flow and resize as needed based on the flow's actual requirements.

Consider enabling accelerated networking on the VMs to increase network performance. For more information, see [Networking for Azure virtual machine scale sets](#).

VM size	vCPU	Memory in GB	Max uncached data disk throughput in I/O operations per second (IOPS) per MBps*	Max network interfaces (NICs) / Expected network bandwidth (Mbps)
Standard_D8s_v3	8	32	12,800/192	4/4,000
Standard_D16s_v3**	16	64	25,600/384	8/8,000
Standard_D32s_v3	32	128	51,200/768	8/16,000
Standard_M16m	16	437.5	10,000/250	8/4,000

\* Disable data disk write caching for all data disks that you use on NiFi nodes.

\*\* We recommend this SKU for most general-purpose data flows. Azure VM SKUs with similar vCPU and memory configurations should also be adequate.

## VM operating system (OS)

We recommend running NiFi in Azure on one of the following guest operating systems:

- Ubuntu 18.04 LTS or later
- CentOS 7.9

To meet the specific requirements of your data flow, it's important to adjust several OS-level settings including:

- Maximum forked processes.
- Maximum file handles.
- The access time, `atime`.

After you adjust the OS to fit your expected use case, use Azure VM Image Builder to codify the generation of those tuned images. For guidance that's specific to NiFi, see [Configuration Best Practices](#) in the Apache NiFi System Administrator's Guide.

## Storage

Store the various NiFi repositories on data disks and not on the OS disk for three main reasons:

- Flows often have high disk throughput requirements that a single disk can't meet.
- It's best to separate the NiFi disk operations from the OS disk operations.
- The repositories shouldn't be on temporary storage.

The following sections outline guidelines for configuring the data disks. These guidelines are specific to Azure. For more information on configuring the repositories, see [State Management](#) in the Apache NiFi System Administrator's Guide.

## Data disk type and size

Consider these factors when you configure the data disks for NiFi:

- Disk type
- Disk size
- Total number of disks

### Note

For up-to-date information on disk types, sizing, and pricing, see [Introduction to Azure managed disks](#).

The following table shows the types of managed disks that are currently available in Azure. You can use NiFi with any of these disk types. But for high-throughput data flows, we recommend Premium SSD.

 [Expand table](#)

	Ultra Disk (NVMe)	Premium SSD	Standard SSD	Standard HDD
<b>Disk type</b>	SSD	SSD	SSD	HDD
<b>Max disk size</b>	65,536 GB	32,767 GB	32,767 GB	32,767 GB
<b>Max throughput</b>	2,000 MiB/s	900 MiB/s	750 MiB/s	500 MiB/s
<b>Max IOPS</b>	160,000	20,000	6,000	2,000

Use at least three data disks to increase throughput of the data flow. For best practices for configuring the repositories on the disks, see [Repository configuration](#) later in this article.

The following table lists the relevant size and throughput numbers for each disk size and type.

[Expand table](#)

	Standard HDD S15	Standard HDD S20	Standard HDD S30	Standard SSD S15	Standard SSD S20	Standard SSD S30	Premium SSD P15	Premium SSD P20	Premium SSD P30
<b>Disk size in GB</b>	256	512	1,024	256	512	1,024	256	512	1,024
<b>IOPS per disk</b>	Up to 500	1,100	2,300	5,000					
<b>Throughput per disk</b>	Up to 60 MBps	125 MBps	150 MBps	200 MBps					

If your system hits VM limits, adding more disks might not increase throughput:

- IOPS and throughput limits depend on the size of the disk.
- The VM size that you choose places IOPS and throughput limits for the VM on all data disks.

For VM-level disk throughput limits, see [Sizes for Linux virtual machines in Azure](#).

## VM disk caching

On Azure VMs, the Host Caching feature manages disk write caching. To increase throughput in data disks that you use for repositories, turn off disk write caching by setting Host Caching to `None`.

## Repository configuration

The best practice guidelines for NiFi are to use a separate disk or disks for each of these repositories:

- Content
- FlowFile
- Provenance

This approach requires a minimum of three disks.

NiFi also supports application-level striping. This functionality increases the size or performance of the data repositories.

The following excerpt is from the `nifi.properties` configuration file. This configuration partitions and stripes the repositories across managed disks that are attached to the VMs:

config

```
nifi.provenance.repository.directory.stripes1=/mnt/disk1/ provenance_repository
nifi.provenance.repository.directory.stripes2=/mnt/disk2/ provenance_repository
nifi.provenance.repository.directory.stripes3=/mnt/disk3/ provenance_repository
nifi.content.repository.directory.stripes1=/mnt/disk4/ content_repository
```

```
nifi.content.repository.directory.stripe2=/mnt/disk5/ content_repository
nifi.content.repository.directory.stripe3=/mnt/disk6/ content_repository
nifi.flowfile.repository.directory=/mnt/disk7/ flowfile_repository
```

For more information about designing for high-performance storage, see [Azure premium storage: design for high performance](#).

## Reporting

NiFi includes a provenance reporting task for the [Log Analytics](#) feature.

You can use this reporting task to offload provenance events to cost-effective, durable long-term storage. The Log Analytics feature provides a [query interface](#) for viewing and graphing the individual events. For more information on these queries, see [Log Analytics queries](#) later in this article.

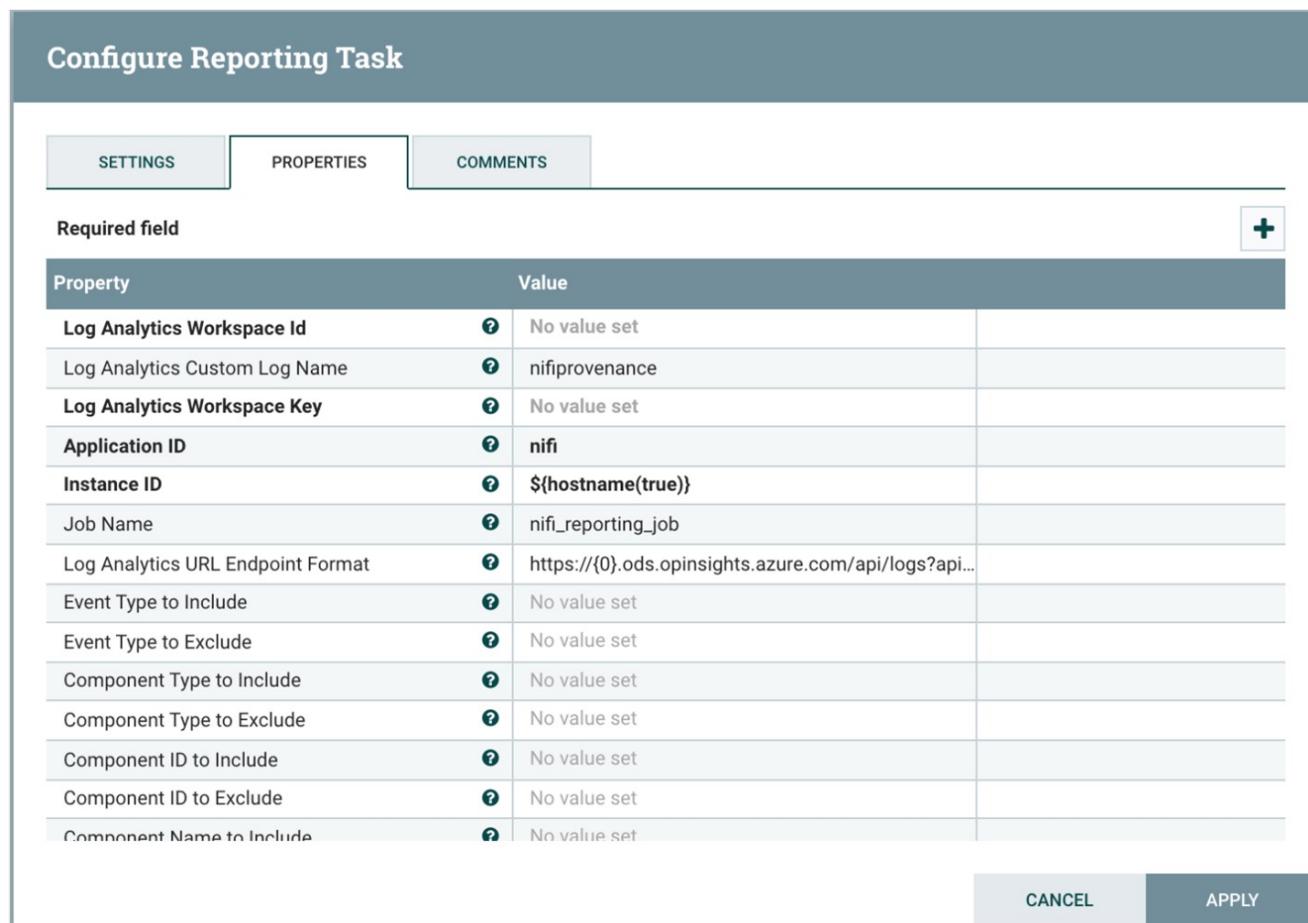
You can also use this task with volatile, in-memory provenance storage. In many scenarios, you can then achieve a throughput increase. But this approach is risky if you need to preserve event data. Ensure that volatile storage meets your durability requirements for provenance events. For more information, see [Provenance Repository](#) in the Apache NiFi System Administrator's Guide.

Before using this process, create a log analytics workspace in your Azure subscription. It's best to set up the workspace in the same region as your workload.

To configure the provenance reporting task:

1. Open the controller settings in NiFi.
2. Select the reporting tasks menu.
3. Select **Create a new reporting task**.
4. Select **Azure Log Analytics Reporting Task**.

The following screenshot shows the properties menu for this reporting task:



Property	Value	
Log Analytics Workspace Id	?	No value set
Log Analytics Custom Log Name	?	nifiprovenance
Log Analytics Workspace Key	?	No value set
Application ID	?	nifi
Instance ID	?	\$(hostname(true))
Job Name	?	nifi_reporting_job
Log Analytics URL Endpoint Format	?	https://{{0}}.ods.opinsights.azure.com/api/logs?api...
Event Type to Include	?	No value set
Event Type to Exclude	?	No value set
Component Type to Include	?	No value set
Component Type to Exclude	?	No value set
Component ID to Include	?	No value set
Component ID to Exclude	?	No value set
Component Name to Include	?	No value set

Two properties are required:

- The log analytics workspace ID
- The log analytics workspace key

You can find these values in the Azure portal by navigating to your Log Analytics workspace.

Other options are also available for customizing and filtering the provenance events that the system sends.

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

You can secure NiFi from an [authentication](#) and [authorization](#) point of view. You can also secure NiFi for all network communication including:

- Within the cluster.
- Between the cluster and ZooKeeper.

See the [Apache NiFi Administrators Guide](#) for instructions on turning on the following options:

- Kerberos
- Lightweight Directory Access Protocol (LDAP)
- Certificate-based authentication and authorization
- Two-way Secure Sockets Layer (SSL) for cluster communications

If you turn on ZooKeeper secure client access, configure NiFi by adding related properties to its `bootstrap.conf` configuration file. The following configuration entries provide an example:

```
config

java.arg.18=-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
java.arg.19=-Dzookeeper.client.secure=true
java.arg.20=-Dzookeeper.ssl.keyStore.location=/path/to/keystore.jks
java.arg.21=-Dzookeeper.ssl.keyStore.password=[KEYSTORE PASSWORD]
java.arg.22=-Dzookeeper.ssl.trustStore.location=/path/to/truststore.jks
java.arg.23=-Dzookeeper.ssl.trustStore.password=[TRUSTSTORE PASSWORD]
```

For general recommendations, see the [Linux security baseline](#).

## Network security

When you implement this solution, keep in mind the following points about network security:

### Network security groups

In Azure, you can use [network security groups](#) to restrict network traffic.

We recommend a *jumpbox* for connecting to the NiFi cluster for administrative tasks. Use this security-hardened VM with [just-in-time \(JIT\) access](#) or [Azure Bastion](#). Set up network security groups to control how you grant access to the jumpbox or Azure Bastion. You can achieve network isolation and control by using network security groups judiciously on the architecture's various subnets.

The following screenshot shows components in a typical virtual network. It contains a common subnet for the jumpbox, virtual machine scale set, and ZooKeeper VMs. This simplified network topology groups components into one subnet. Follow your organization's guidelines for separation of duties and network design.

Device	Type	Subnet
jumpbox-nic	Network interface	subnet-nifi
nifi-test-flow-vmss (instance 0)	Scale set instance	subnet-nifi
nifi-test-flow-vmss (instance 1)	Scale set instance	subnet-nifi
nifi-test-flow-vmss (instance 2)	Scale set instance	subnet-nifi
nifi-test-flow-zk-nic-0	Network interface	subnet-nifi
nifi-test-flow-zk-nic-1	Network interface	subnet-nifi
nifi-test-flow-zk-nic-2	Network interface	subnet-nifi

### Outbound internet access consideration

NiFi in Azure doesn't need access to the public internet to run. If the data flow doesn't need internet access to retrieve data, improve the cluster's security by following these steps to disable outbound internet access:

1. Create an additional network security group rule in the virtual network.

2. Use these settings:

- Source: Any
- Destination: Internet
- Action: Deny

Outbound security rules						
Priority	Name	Port	Protocol	Source	Destination	Action
110	DenyInternet	Any	Any	VirtualNetwork	Internet	 Allow

With this rule in place, you can still access some Azure services from the data flow if you configure a private endpoint in the virtual network. Use [Azure Private Link](#) for this purpose. This service provides a way for your traffic to travel on the Microsoft backbone network while not requiring any other external network access. NiFi currently supports Private Link for the Blob Storage and Data Lake Storage processors. If a network time protocol (NTP) server isn't available in your private network, allow outbound access to NTP. For detailed information, see [Time sync for Linux VMs in Azure](#).

## Data protection

It's possible to operate NiFi unsecured, without wire encryption, identity and access management (IAM), or data encryption. But it's best to secure production and public-cloud deployments in these ways:

- Encrypting communication with Transport Layer Security (TLS)
- Using a supported authentication and authorization mechanism
- Encrypting data at rest

Azure Storage provides server-side transparent data encryption. But starting with the 1.13.2 release, NiFi doesn't configure wire encryption or IAM by default. This behavior might change in future releases.

The following sections show how to secure deployments in these ways:

- Enable wire encryption with TLS
- Configure authentication that's based on certificates or Microsoft Entra ID
- Manage encrypted storage on Azure

## Disk encryption

To improve security, use Azure disk encryption. For a detailed procedure, see [Encrypt OS and attached data disks in a virtual machine scale set with the Azure CLI](#). That document also contains instructions on providing your own encryption key. The following steps outline a basic example for NiFi that work for most deployments:

1. To turn on disk encryption in an existing Key Vault instance, use the following Azure CLI command:

```
Azure CLI
az keyvault create --resource-group myResourceGroup --name myKeyVaultName --enabled-for-disk-encryption
```

2. Turn on encryption of the virtual machine scale set data disks with the following command:

```
Azure CLI
az vmss encryption enable --resource-group myResourceGroup --name myScaleSet --disk-encryption-keyvault myKeyVaultID
--volume-type DATA
```

3. You can optionally use a key encryption key (KEK). Use the following Azure CLI command to encrypt with a KEK:

```
Azure CLI
az vmss encryption enable --resource-group myResourceGroup --name myScaleSet \
--disk-encryption-keyvault myKeyVaultID \
--key-encryption-keyvault myKeyVaultID \
--key-encryption-key https://<mykeyvaultname>.vault.azure.net/keys/myKey/<version> \
--volume-type DATA
```

### ⓘ Note

If you configured your virtual machine scale set for manual update mode, run the `update-instances` command. Include the version of the encryption key that you stored in Key Vault.

## Encryption in transit

NiFi supports TLS 1.2 for encryption in transit. This protocol offers protection for user access to the UI. With clusters, the protocol protects communication between NiFi nodes. It can also protect communication with ZooKeeper. When you enable TLS, NiFi uses mutual TLS (mTLS) for mutual authentication for:

- Certificate-based client authentication if you configured this type of authentication.
- All intracluster communication.

To enable TLS, take the following steps:

1. Create a keystore and a truststore for client–server and intracluster communication and authentication.
2. Configure `$NIFI_HOME/conf/nifi.properties`. Set the following values:
  - Hostnames
  - Ports
  - Keystore properties
  - Truststore properties
  - Cluster and ZooKeeper security properties, if applicable
3. Configure authentication in `$NIFI_HOME/conf/authorizers.xml`, typically with an initial user that has certificate-based authentication or another option.
4. Optionally configure mTLS and a proxy read policy between NiFi and any proxies, load balancers, or external endpoints.

For a complete walkthrough, see [Securing NiFi with TLS](#) in the Apache project documentation.

### ⓘ Note

As of version 1.13.2:

- NiFi doesn't enable TLS by default.
- There's no out-of-the-box support for anonymous and single user access for TLS-enabled NiFi instances.

To enable TLS for encryption in transit, configure a user group and policy provider for authentication and authorization in `$NIFI_HOME/conf/authorizers.xml`. For more information, see [Identity and access control](#) later in this article.

## Certificates, keys, and keystores

To support TLS, generate certificates, store them in Java KeyStore and TrustStore, and distribute them across a NiFi cluster. There are two general options for certificates:

- Self-signed certificates
- Certificates that certified authorities (CAs) sign

With CA-signed certificates, it's best to use an intermediate CA to generate certificates for nodes in the cluster.

KeyStore and TrustStore are the key and certificate containers in the Java platform. KeyStore stores the private key and certificate of a node in the cluster. TrustStore stores one of the following types of certificates:

- All trusted certificates, for self-signed certificates in KeyStore
- A certificate from a CA, for CA-signed certificates in KeyStore

Keep the scalability of your NiFi cluster in mind when you choose a container. For instance, you might want to increase or decrease the number of nodes in a cluster in the future. Choose CA-signed certificates in KeyStore and one or more certificates from a CA in TrustStore in that case. With this option, there's no need to update the existing TrustStore in the existing nodes of the cluster. An existing TrustStore trusts and accepts certificates from these types of nodes:

- Nodes that you add to the cluster
- Nodes that replace other nodes in the cluster

## NiFi configuration

To enable TLS for NiFi, use `$NIFI_HOME/conf/nifi.properties` to configure the properties in this table. Ensure that the following properties include the hostname that you use to access NiFi:

- `nifi.web.https.host` or `nifi.web.proxy.host`
- The host certificate's designated name or subject alternative names

Otherwise, a hostname verification failure or an HTTP HOST header verification failure might result, denying you access.

 [Expand table](#)

Property name	Description	Example values
<code>nifi.web.https.host</code>	Hostname or IP address to use for the UI and REST API. This value should be internally resolvable. We recommend not using a publicly accessible name.	<code>nifi.internal.cloudapp.net</code>
<code>nifi.web.https.port</code>	HTTPS port to use for the UI and REST API.	<code>9443</code> (default)
<code>nifi.web.proxy.host</code>	Comma-separated list of alternate hostnames that clients use to access the UI and REST API. This list typically includes any hostname that's specified as a subject alternative name (SAN) in the server certificate. The list can also include any hostname and port that a load balancer, proxy, or Kubernetes ingress controller uses.	<code>40.67.218.235, 40.67.218.235:443, nifi.westus2.cloudapp.com, nifi.westus2.cloudapp.com:443</code>
<code>nifi.security.keystore</code>	The path to a JKS or PKCS12 keystore that contains the certificate's private key.	<code>./conf/keystore.jks</code>
<code>nifi.security.keystoreType</code>	The keystore type.	<code>JKS</code> or <code>PKCS12</code>
<code>nifi.security.keystorePasswd</code>	The keystore password.	<code>08SitLBYPz7g/RpsqH+zM</code>
<code>nifi.security.keyPasswd</code>	(Optional) The password for the private key.	
<code>nifi.security.truststore</code>	The path to a JKS or PKCS12 truststore that contains certificates or CA certificates that authenticate trusted users and cluster nodes.	<code>./conf/truststore.jks</code>
<code>nifi.security.truststoreType</code>	The truststore type.	<code>JKS</code> or <code>PKCS12</code>
<code>nifi.security.truststorePasswd</code>	The truststore password.	<code>RJ1pGe6/TuN5fG+VnaEPI</code>
<code>nifi.cluster.protocol.is.secure</code>	The status of TLS for intracluster communication. If <code>nifi.cluster.is.node</code> is <code>true</code> , set this value to <code>true</code> to enable cluster TLS.	<code>true</code>
<code>nifi.remote.input.secure</code>	The status of TLS for site-to-site communication.	<code>true</code>

The following example shows how these properties appear in `$NIFI_HOME/conf/nifi.properties`. Note that the `nifi.web.http.host` and `nifi.web.http.port` values are blank.

config

```

nifi.remote.input.secure=true
nifi.web.http.host=
nifi.web.http.port=
nifi.web.https.host=nifi.internal.cloudapp.net

```

```

nifi.web.https.port=9443
nifi.web.proxy.host=40.67.218.235, 40.67.218.235:443, nifi.westus2.cloudapp.com, nifi.westus2.cloudapp.com:443
nifi.security.keystore=./conf/keystore.jks
nifi.security.keystoreType=JKS
nifi.security.keystorePasswd=08SitLBypCz7g/RpsqH+zM
nifi.security.keyPasswd=
nifi.security.truststore=./conf/truststore.jks
nifi.security.truststoreType=JKS
nifi.security.truststorePasswd=RJlpGe6/TuN5fG+VnaEPi8
nifi.cluster.protocol.is.secure=true

```

## ZooKeeper configuration

For instructions on [enabling TLS in Apache ZooKeeper](#) for quorum communications and client access, see the [ZooKeeper Administrator's Guide](#). Only versions 3.5.5 or later support this functionality.

NiFi uses ZooKeeper for its zero-leader clustering and cluster coordination. Starting with version 1.13.0, NiFi supports secure client access to TLS-enabled instances of ZooKeeper. ZooKeeper stores cluster membership and cluster-scoped processor state in plain text. So it's important to use secure client access to ZooKeeper to authenticate ZooKeeper client requests. Also encrypt sensitive values in transit.

To enable TLS for NiFi client access to ZooKeeper, set the following properties in `$NIFI_HOME/conf/nifi.properties`. If you set `nifi.zookeeper.client.secure true` without configuring `nifi.zookeeper.security` properties, NiFi falls back to the keystore and truststore that you specify in `nifi.securityproperties`.

 [Expand table](#)

Property name	Description	Example values
<code>nifi.zookeeper.client.secure</code>	The status of client TLS when connecting to ZooKeeper.	<code>true</code>
<code>nifi.zookeeper.security.keystore</code>	The path to a JKS, PKCS12, or PEM keystore that contains the private key of the certificate that's presented to ZooKeeper for authentication.	<code>./conf/zookeeper.keystore.jks</code>
<code>nifi.zookeeper.security.keystoreType</code>	The keystore type.	<code>JKS</code> , <code>PKCS12</code> , <code>PEM</code> , or autodetect by extension
<code>nifi.zookeeper.security.keystorePasswd</code>	The keystore password.	<code>caB6ECKi03R/co+N+641rz</code>
<code>nifi.zookeeper.security.keyPasswd</code>	(Optional) The password for the private key.	
<code>nifi.zookeeper.security.truststore</code>	The path to a JKS, PKCS12, or PEM truststore that contains certificates or CA certificates that are used to authenticate ZooKeeper.	<code>./conf/zookeeper.truststore.jks</code>
<code>nifi.zookeeper.security.truststoreType</code>	The truststore type.	<code>JKS</code> , <code>PKCS12</code> , <code>PEM</code> , or autodetect by extension
<code>nifi.zookeeper.security.truststorePasswd</code>	The truststore password.	<code>qBdnLhsp+mKvV7wab/L4sv</code>
<code>nifi.zookeeper.connect.string</code>	The connection string to the ZooKeeper host or quorum. This string is a comma-separated list of <code>host:port</code> values. Typically the <code>secureClientPort</code> value isn't the same as the <code>clientPort</code> value. See your ZooKeeper configuration for the correct value.	<code>zookeeper1.internal.cloudapp.net:2281, zookeeper2.internal.cloudapp.net:2281, zookeeper3.internal.cloudapp.net:2281</code>

The following example shows how these properties appear in `$NIFI_HOME/conf/nifi.properties`:

config

```
nifi.zookeeper.client.secure=true
nifi.zookeeper.security.keystore=../conf/keystore.jks
nifi.zookeeper.security.keystoreType=JKS
nifi.zookeeper.security.keystorePasswd=caB6ECKi03R/co+N+64lrz
nifi.zookeeper.security.keyPasswd=
nifi.zookeeper.security.truststore=../conf/truststore.jks
nifi.zookeeper.security.truststoreType=JKS
nifi.zookeeper.security.truststorePasswd=qBdnLhsp+mKvV7wab/L4sv
nifi.zookeeper.connect.string=zookeeper1.internal.cloudapp.net:2281,zookeeper2.internal.cloudapp.net:2281,zookeeper3.internal.cloudapp.net:2281
```

For more information about securing ZooKeeper with TLS, see the [Apache NiFi Administration Guide](#).

## Identity and access control

In NiFi, identity and access control is achieved through user authentication and authorization. For user authentication, NiFi has multiple options to choose from: Single User, LDAP, Kerberos, Security Assertion Markup Language (SAML), and OpenID Connect (OIDC). If you don't configure an option, NiFi uses client certificates to authenticate users over HTTPS.

If you're considering multifactor authentication, we recommend the combination of Microsoft Entra ID and [OIDC](#). Microsoft Entra ID supports cloud-native single sign-on (SSO) with OIDC. With this combination, users can take advantage of many enterprise security features:

- Logging and alerting on suspicious activities from user accounts
- Monitoring attempts to access deactivated credentials
- Alerting on unusual account sign-in behavior

For authorization, NiFi provides enforcement that's based on user, group, and access policies. NiFi provides this enforcement through UserGroupProviders and AccessPolicyProviders. By default, providers include File, LDAP, Shell, and Azure Graph-based UserGroupProviders. With [AzureGraphUserGroupProvider](#), you can source user groups from Microsoft Entra ID. You can then assign policies to these groups. For configuration instructions, see the [Apache NiFi Administration Guide](#).

AccessPolicyProviders that are based on files and Apache Ranger are currently available for managing and storing user and group policies. For detailed information, see the [Apache NiFi documentation](#) and [Apache Ranger documentation](#).

## Application gateway

An application gateway provides a managed layer-7 load balancer for the NiFi interface. Configure the application gateway to use the virtual machine scale set of the NiFi nodes as its back-end pool.

For most NiFi installations, we recommend the following [Application Gateway](#) configuration:

- Tier: Standard
- SKU size: medium
- Instance count: two or more

Use a [health probe](#) to monitor the health of the web server on each node. Remove unhealthy nodes from the load balancer rotation. This approach makes it easier to view the user interface when the overall cluster is unhealthy. The browser only directs you to nodes that are currently healthy and responding to requests.

There are two key health probes to consider. Together they provide a regular heartbeat on the overall health of every node in the cluster. Configure the first health probe to point to the path `/NiFi`. This probe determines the health of the NiFi user interface on each node. Configure a second health probe for the path `/nifi-api/controller/cluster`. This probe indicates whether each node is currently healthy and joined to the overall cluster.

You have two options for configuring the application gateway's front-end IP address:

- With a public IP address
- With a private subnet IP address

Only include a public IP address if users need to access the UI over the public internet. If public internet access for users isn't required, access the load balancer front end from a jumpbox in the virtual network or through peering to your private network. If you configure the application gateway with a public IP address, we recommend enabling client certificate authentication for NiFi and enabling TLS for the NiFi UI. You can also use a network security group in the delegated application gateway subnet to limit source IP addresses.

## Diagnostics and health monitoring

Within the diagnostics settings of Application Gateway, there's a configuration option for sending metrics and access logs. By using this option, you can send this information from the load balancer to various places:

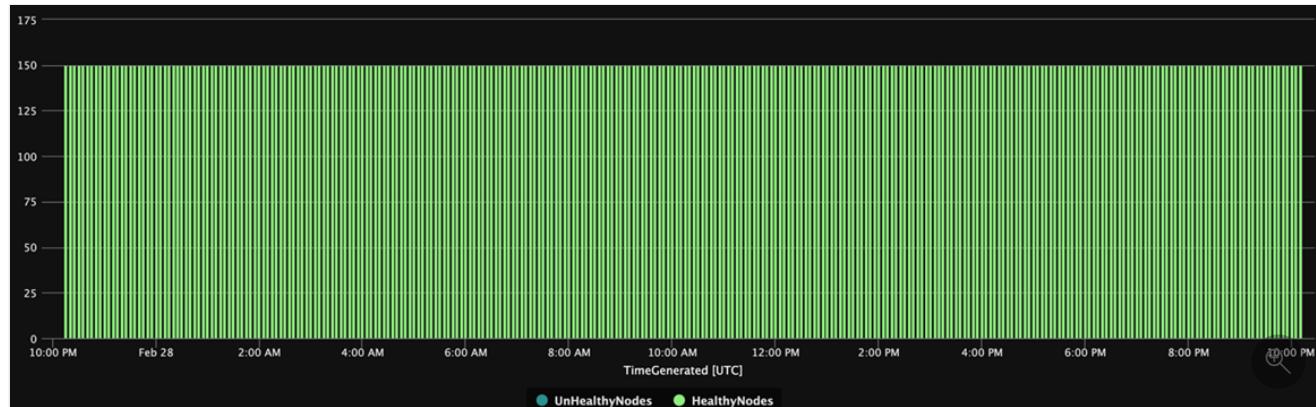
- A storage account
- Event Hubs
- A Log Analytics workspace

Turning on this setting is useful for debugging load-balancing issues and for gaining insight into the health of cluster nodes.

The following [Log Analytics](#) query shows cluster node health over time from an Application Gateway perspective. You can use a similar query to generate alerts or automated repair actions for unhealthy nodes.

```
Kusto
AzureDiagnostics
| summarize UnHealthyNodes = max(unHealthyHostCount_d), HealthyNodes = max(healthyHostCount_d) by bin(TimeGenerated, 5m)
| render timechart
```

The following chart of the query results shows a time view of the health of the cluster:



## Availability

When you implement this solution, keep in mind the following points about availability:

### Load balancer

Use a load balancer for the UI to increase UI availability during node downtime.

### Separate VMs

To increase availability, deploy the ZooKeeper cluster on separate VMs from the VMs in the NiFi cluster. For more information on configuring ZooKeeper, see [State Management](#) in the Apache NiFi System Administrator's Guide.

### Availability zones

Deploy both the NiFi virtual machine scale set and the ZooKeeper cluster in a cross-zone configuration to maximize availability. When communication between the nodes in the cluster crosses availability zones, it introduces a small amount of latency. But this latency typically has a minimal overall effect on the throughput of the cluster.

### Virtual machine scale sets

We recommend deploying the NiFi nodes into a single virtual machine scale set that spans availability zones where available. For detailed information on using scale sets in this way, see [Create a virtual machine scale set that uses Availability Zones](#).

## Monitoring

Multiple options are available for monitoring the health and performance of a NiFi cluster:

- Reporting tasks.
- [MonitoFi](#), a separate Microsoft-developed application. MonitoFi runs externally and monitors the cluster by using the NiFi API.

## Reporting task-based monitoring

For monitoring, you can use a reporting task that you configure and run in NiFi. As [Diagnostics and health monitoring](#) discusses, Log Analytics provides a reporting task in the NiFi Azure bundle. You can use that reporting task to integrate the monitoring with Log Analytics and existing monitoring or logging systems.

## Log Analytics queries

Sample queries in the following sections can help you get started. For an overview of how to query Log Analytics data, see [Azure Monitor log queries](#).

Log queries in Monitor and Log Analytics use a version of the [Kusto query language](#). But differences exist between log queries and Kusto queries. For more information, see [Kusto query overview](#).

For more structured learning, see these tutorials:

- [Get started with log queries in Azure Monitor](#)
- [Get started with Log Analytics in Azure Monitor](#)

## Log Analytics reporting task

By default, NiFi sends metrics data to the `nifimetrics` table. But you can configure a different destination in the reporting task properties. The reporting task captures the following NiFi metrics:

 [Expand table](#)

Metric type	Metric name
NiFi Metrics	<code>FlowFilesReceived</code>
NiFi Metrics	<code>FlowFilesSent</code>
NiFi Metrics	<code>FlowFilesQueued</code>
NiFi Metrics	<code>BytesReceived</code>
NiFi Metrics	<code>BytesWritten</code>
NiFi Metrics	<code>BytesRead</code>
NiFi Metrics	<code>BytesSent</code>
NiFi Metrics	<code>BytesQueued</code>
Port status metrics	<code>InputCount</code>
Port status metrics	<code>InputBytes</code>
Connection status metrics	<code>QueuedCount</code>
Connection status metrics	<code>QueuedBytes</code>
Port status metrics	<code>OutputCount</code>
Port status metrics	<code>OutputBytes</code>

Metric type	Metric name
JVM Metrics	jvm.uptime
JVM Metrics	jvm.heap_used
JVM Metrics	jvm.heap_usage
JVM Metrics	jvm.non_heap_usage
JVM Metrics	jvm.thread_states.runnable
JVM Metrics	jvm.thread_states.blocked
JVM Metrics	jvm.thread_states.timed_waiting
JVM Metrics	jvm.thread_states.terminated
JVM Metrics	jvm.thread_count
JVM Metrics	jvm.daemon_thread_count
JVM Metrics	jvm.file_descriptor_usage
JVM Metrics	jvm.gc.runs jvm.gc.runs.g1_old_generation jvm.gc.runs.g1_young_generation
JVM Metrics	jvm.gc.time jvm.gc.time.g1_young_generation jvm.gc.time.g1_old_generation
JVM Metrics	jvm.buff_pool_direct_capacity
JVM Metrics	jvm.buff_pool_direct_count
JVM Metrics	jvm.buff_pool_direct_mem_used
JVM Metrics	jvm.buff_pool_mapped_capacity
JVM Metrics	jvm.buff_pool_mapped_count
JVM Metrics	jvm.buff_pool_mapped_mem_used
JVM Metrics	jvm.mem_pool_code_cache
JVM Metrics	jvm.mem_pool_compressed_class_space
JVM Metrics	jvm.mem_pool_g1_eden_space
JVM Metrics	jvm.mem_pool_g1_old_gen
JVM Metrics	jvm.mem_pool_g1_survivor_space
JVM Metrics	jvm.mem_pool_metaspace
JVM Metrics	jvm.thread_states.new
JVM Metrics	jvm.thread_states.waiting

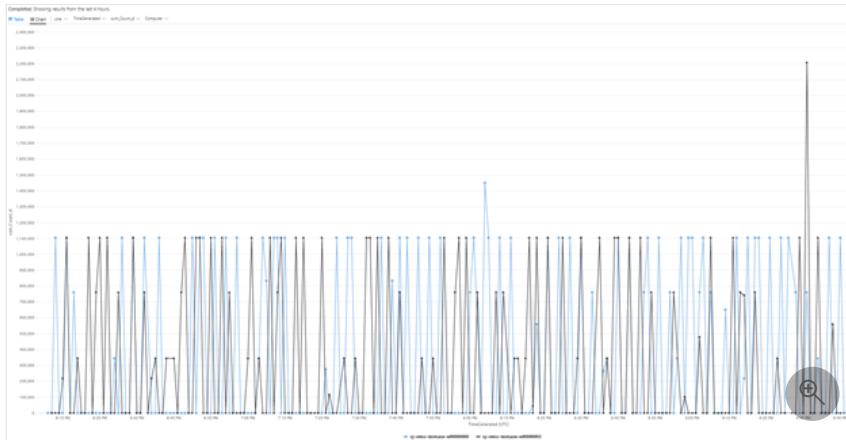
Metric type	Metric name
Processor Level metrics	BytesRead
Processor Level metrics	BytesWritten
Processor Level metrics	FlowFilesReceived
Processor Level metrics	FlowFilesSent

Here's a sample query for the `BytesQueued` metric of a cluster:

Kusto

```
let table_name = nifimetrics_CL;
let metric = "BytesQueued";
table_name
| where Name_s == metric
| where Computer contains {ComputerName}
| project TimeGenerated, Computer, ProcessGroupName_s, Count_d, Name_s
| summarize sum(Count_d) by bin(TimeGenerated, 1m), Computer, Name_s
| render timechart
```

That query produces a chart like the one in this screenshot:



#### ⚠ Note

When you run NiFi on Azure, you're not limited to the Log Analytics reporting task. NiFi supports reporting tasks for many third-party monitoring technologies. For a list of supported reporting tasks, see the [Reporting Tasks](#) section of the [Apache NiFi Documentation index](#).

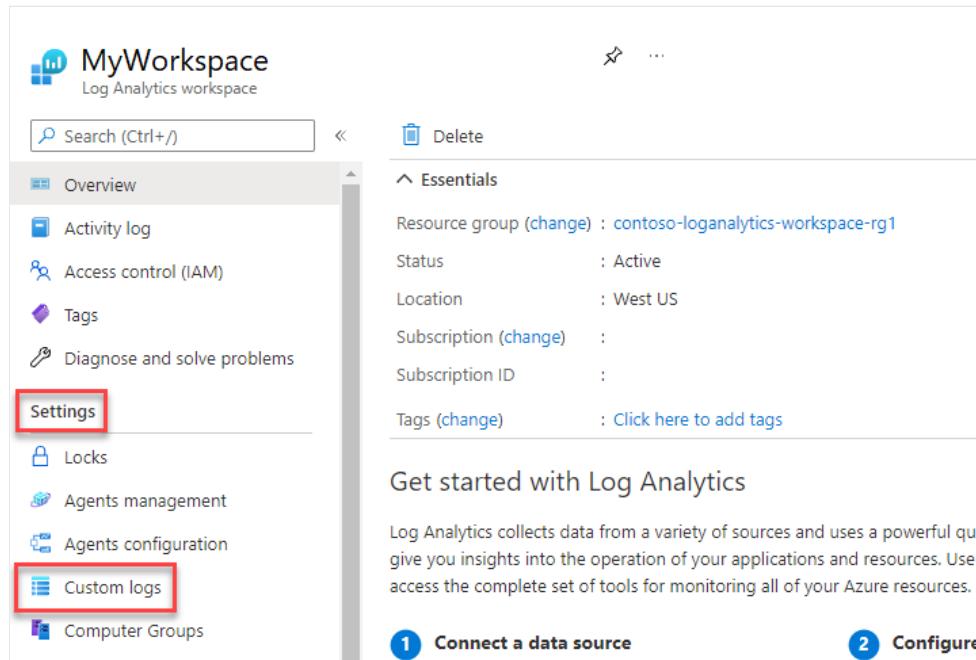
## NiFi infrastructure monitoring

Besides the reporting task, install the [Log Analytics VM extension](#) on the NiFi and ZooKeeper nodes. This extension gathers logs, additional VM-level metrics, and metrics from ZooKeeper.

## Custom logs for the NiFi app, user, bootstrap, and ZooKeeper

To capture more logs, follow these steps:

1. In the Azure portal, select **Log Analytics workspaces**, and then select your workspace.
2. Under **Settings**, select **Custom logs**.



MyWorkspace  
Log Analytics workspace

Search (Ctrl+ /) Overview Delete

Activity log  
Access control (IAM)  
Tags  
Diagnose and solve problems

**Settings**

Locks  
Agents management  
Agents configuration  
**Custom logs**

Computer Groups

**Essentials**

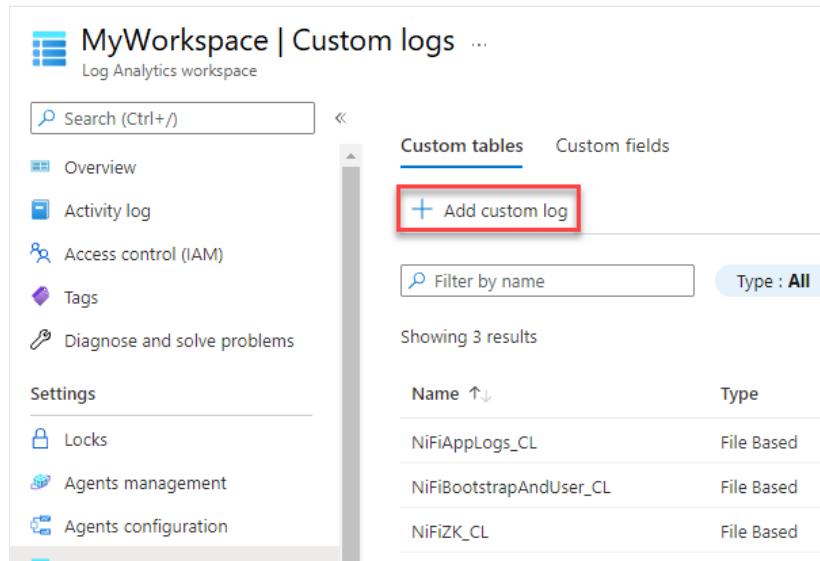
Resource group (change) : contoso-loganalytics-workspace-rg1  
Status : Active  
Location : West US  
Subscription (change) :  
Subscription ID :

Get started with Log Analytics

Log Analytics collects data from a variety of sources and uses a powerful query language to give you insights into the operation of your applications and resources. Use the Log Analytics workspace to access the complete set of tools for monitoring all of your Azure resources.

1 Connect a data source 2 Configure

3. Select Add custom log.



MyWorkspace | Custom logs

Search (Ctrl+ /) Overview Custom tables Custom fields

Activity log  
Access control (IAM)  
Tags  
Diagnose and solve problems

Settings

Locks  
Agents management  
Agents configuration

**+ Add custom log**

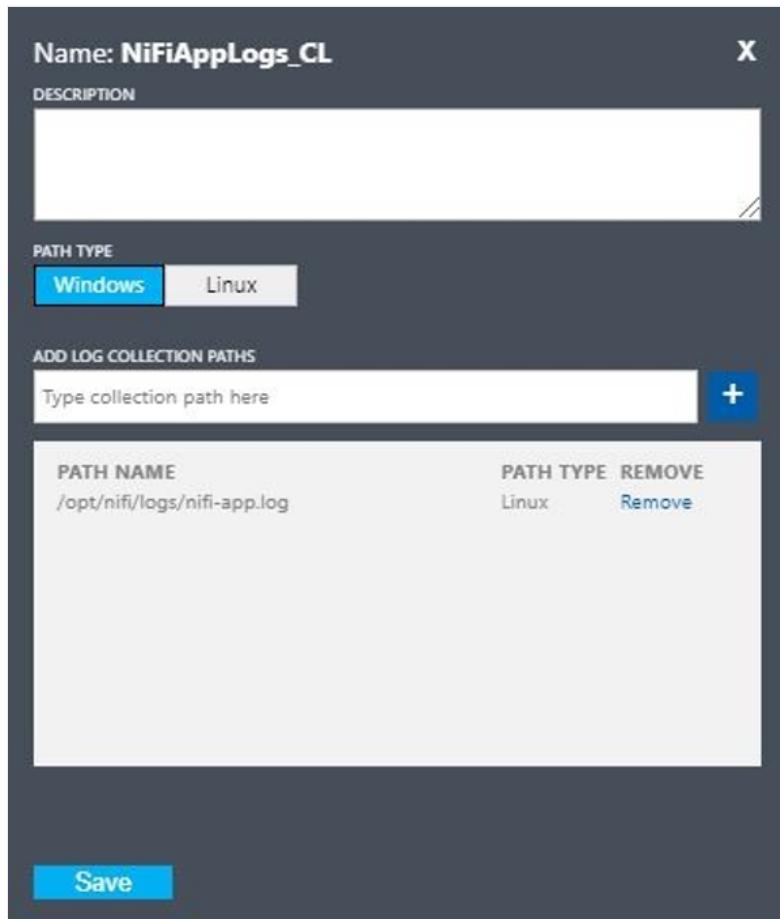
Filter by name Type : All

Showing 3 results

Name	Type
NiFiAppLogs_CL	File Based
NiFiBootstrapAndUser_CL	File Based
NiFiZK_CL	File Based

4. Set up a custom log with these values:

- Name: NiFiAppLogs
- Path type: Linux
- Path name: /opt/nifi/logs/nifi-app.log



5. Set up a custom log with these values:

- Name: NiFiBootstrapAndUser
- First path type: Linux
- First path name: /opt/nifi/logs/nifi-user.log
- Second path type: Linux
- Second path name: /opt/nifi/logs/nifi-bootstrap.log

Name: **NiFiBootstrapAndUser\_CL**

DESCRIPTION

PATH TYPE

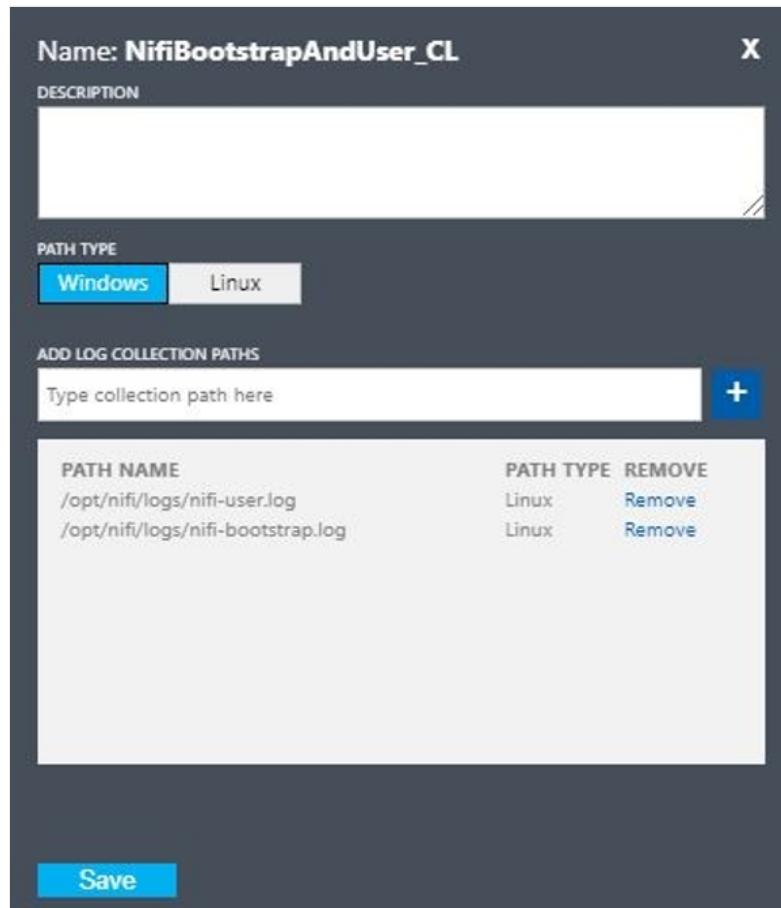
**Windows**   Linux

ADD LOG COLLECTION PATHS

Type collection path here **+**

PATH NAME	PATH TYPE	REMOVE
/opt/nifi/logs/nifi-user.log	Linux	Remove
/opt/nifi/logs/nifi-bootstrap.log	Linux	Remove

**Save**



6. Set up a custom log with these values:

- Name: **NiFiZK**
- Path type: **Linux**
- Path name: **/opt/zookeeper/logs/\*.out**

Name: **NiFiZK\_CL**

DESCRIPTION

PATH TYPE

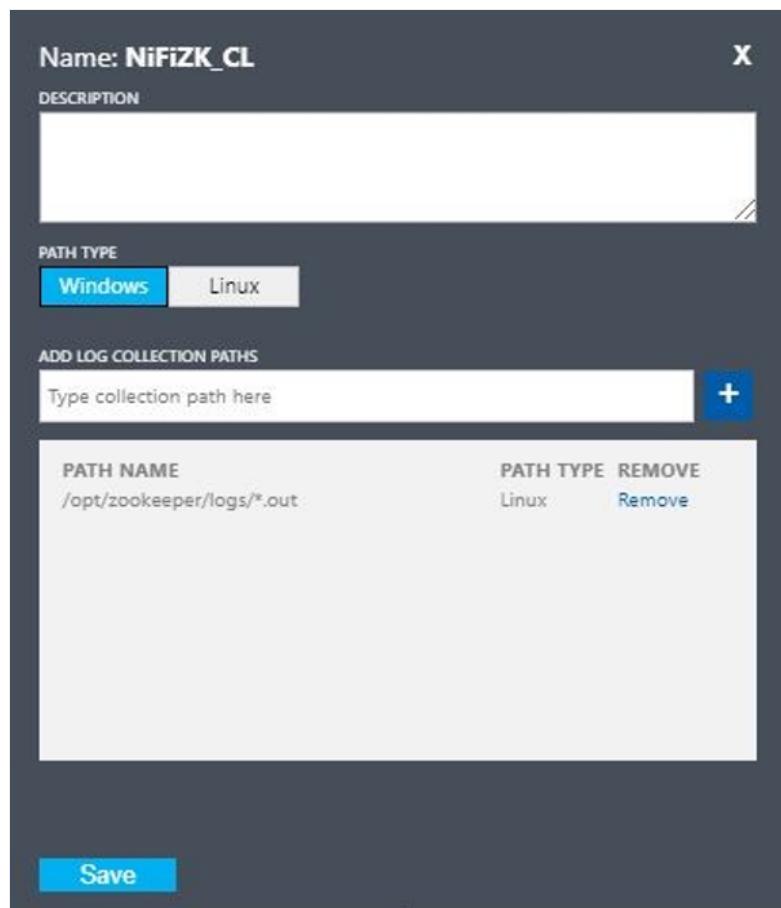
**Windows**   Linux

ADD LOG COLLECTION PATHS

Type collection path here **+**

PATH NAME	PATH TYPE	REMOVE
/opt/zookeeper/logs/*.out	Linux	Remove

**Save**



Here's a sample query of the `NiFiAppLogs` custom table that the first example created:

```
Kusto

NiFiAppLogs_CL
| where TimeGenerated > ago(24h)
| where Computer contains {ComputerName} and RawData contains "error"
| limit 10
```

That query produces results similar to the following results:

TimeGenerated [UTC]	Computer	RawData	Type	ResourceId
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,887 ERROR [Index Provenance Events-1] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,887 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,887 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-1] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-1] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20
2/27/2020, 10:31:06.000 PM	nifi-longhaul-nifi00000N	2020-02-27 04:51:41,888 ERROR [Index Provenance Events-2] o.a.n.p.i...	NiFiAppLogs_CL	/subscriptions/a5ab90eb-20

## Infrastructure log configuration

You can use Monitor to monitor and manage VMs or physical computers. These resources can be in your local datacenter or other cloud environment. To set up this monitoring, deploy the Log Analytics agent. Configure the agent to report to a Log Analytics workspace. For more information, see [Log Analytics agent overview](#).

The following screenshot shows a sample agent configuration for NiFi VMs. The `Perf` table stores the collected data.

Here's a sample query for the NiFi app `Perf` logs:

```
Kusto

let cluster_name = {ComputerName};
// The hourly average of CPU usage across all computers.
Perf
| where Computer contains {ComputerName}
| where CounterName == "% Processor Time" and InstanceName == "_Total"
| where ObjectName == "Processor"
| summarize CPU_Time_Avg = avg(CounterValue) by bin(TimeGenerated, 30m), Computer
```

That query produces a report like the one in this screenshot:



## Alerts

Use Monitor to create alerts on the health and performance of the NiFi cluster. Example alerts include:

- The total queue count has exceeded a threshold.
- The `BytesWritten` value is under an expected threshold.
- The `FlowFilesReceived` value is under a threshold.
- The cluster is unhealthy.

For more information on setting up alerts in Monitor, see [Overview of alerts in Microsoft Azure](#).

## Configuration parameters

The following sections discuss recommended, non-default configurations for NiFi and its dependencies, including ZooKeeper and Java. These settings are suited for cluster sizes that are possible in the cloud. Set the properties in the following configuration files:

- `$NIFI_HOME/conf/nifi.properties`
- `$NIFI_HOME/conf/bootstrap.conf`
- `$ZOOKEEPER_HOME/conf/zoo.cfg`
- `$ZOOKEEPER_HOME/bin/zkEnv.sh`

For detailed information about available configuration properties and files, see the [Apache NiFi System Administrator's Guide](#) and [ZooKeeper Administrator's Guide](#).

## NiFi

For an Azure deployment, consider adjusting properties in `$NIFI_HOME/conf/nifi.properties`. The following table lists the most important properties. For more recommendations and insights, see the [Apache NiFi mailing lists](#).

[Expand table](#)

Parameter	Description	Default	Recommendation
<code>nifi.cluster.node.connection.timeout</code>	How long to wait when opening a connection to other cluster nodes.	5 seconds	60 seconds
<code>nifi.cluster.node.read.timeout</code>	How long to wait for a response when making a request to other cluster nodes.	5 seconds	60 seconds
<code>nifi.cluster.protocol.heartbeat.interval</code>	How often to send heartbeats back to the cluster coordinator.	5 seconds	60 seconds
<code>nifi.cluster.node.max.concurrent.requests</code>	What level of parallelism to use when replicating HTTP calls like REST API calls to other cluster nodes.	100	500
<code>nifi.cluster.node.protocol.threads</code>	Initial thread pool size for inter-cluster/replicated communications.	10	50
<code>nifi.cluster.node.protocol.max.threads</code>	Maximum number of threads to use for inter-	50	75

Parameter	Description	Default	Recommendation
	cluster/replicated communications.		
<code>nifi.cluster.flow.election.max.candidates</code>	Number of nodes to use when deciding what the current flow is. This value short-circuits the vote at the specified number.	empty	75
<code>nifi.cluster.flow.election.max.wait.time</code>	How long to wait on nodes before deciding what the current flow is.	5 minutes	5 minutes

## Cluster behavior

When you configure clusters, keep in mind the following points.

### Timeout

To ensure the overall health of a cluster and its nodes, it can be beneficial to increase timeouts. This practice helps guarantee that failures don't result from transient network problems or high loads.

In a distributed system, the performance of individual systems varies. This variation includes network communications and latency, which usually affects inter-node, inter-cluster communication. The network infrastructure or the system itself can cause this variation. As a result, the probability of variation is very likely in large clusters of systems. In Java applications under load, pauses in garbage collection (GC) in the Java virtual machine (JVM) can also affect request response times.

Use properties in the following sections to configure timeouts to suit your system's needs:

#### `nifi.cluster.node.connection.timeout` and `nifi.cluster.node.read.timeout`

The `nifi.cluster.node.connection.timeout` property specifies how long to wait when opening a connection. The `nifi.cluster.node.read.timeout` property specifies how long to wait when receiving data between requests. The default value for each property is five seconds. These properties apply to node-to-node requests. Increasing these values helps alleviate several related problems:

- Being disconnected by the cluster coordinator because of heartbeat interruptions
- Failure to get the flow from the coordinator when joining the cluster
- Establishing site-to-site (S2S) and load-balancing communications

Unless your cluster has a very small scale set, such as three nodes or fewer, use values that are greater than the defaults.

#### `nifi.cluster.protocol.heartbeat.interval`

As part of the NiFi clustering strategy, each node emits a heartbeat to communicate its healthy status. By default, nodes send heartbeats every five seconds. If the cluster coordinator detects that eight heartbeats in a row from a node have failed, it disconnects the node. Increase the interval that's set in the `nifi.cluster.protocol.heartbeat.interval` property to help accommodate slow heartbeats and prevent the cluster from disconnecting nodes unnecessarily.

### Concurrency

Use properties in the following sections to configure concurrency settings:

#### `nifi.cluster.node.protocol.threads` and `nifi.cluster.node.protocol.max.threads`

The `nifi.cluster.node.protocol.max.threads` property specifies the maximum number of threads to use for all-cluster communications such as S2S load balancing and UI aggregation. The default for this property is 50 threads. For large clusters, increase this value to account for the greater number of requests that these operations require.

The `nifi.cluster.node.protocol.threads` property determines the initial thread pool size. The default value is 10 threads. This value is a minimum. It grows as needed up to the maximum set in `nifi.cluster.node.protocol.max.threads`. Increase the `nifi.cluster.node.protocol.threads` value for clusters that use a large scale set at launch.

## **nifi.cluster.node.max.concurrent.requests**

Many HTTP requests like REST API calls and UI calls need to be replicated to other nodes in the cluster. As the size of the cluster grows, an increasing number of requests get replicated. The `nifi.cluster.node.max.concurrent.requests` property limits the number of outstanding requests. Its value should exceed the expected cluster size. The default value is 100 concurrent requests. Unless you're running a small cluster of three or fewer nodes, prevent failed requests by increasing this value.

## **Flow election**

Use properties in the following sections to configure flow election settings:

### **nifi.cluster.flow.election.max.candidates**

NiFi uses zero-leader clustering, which means there isn't one specific authoritative node. As a result, nodes vote on which flow definition counts as the correct one. They also vote to decide which nodes join the cluster.

By default, the `nifi.cluster.flow.election.max.candidates` property is the maximum wait time that the `nifi.cluster.flow.election.max.wait.time` property specifies. When this value is too high, startup can be slow. The default value for `nifi.cluster.flow.election.max.wait.time` is five minutes. Set the maximum number of candidates to a non-empty value like `1` or greater to ensure that the wait is no longer than needed. If you set this property, assign it a value that corresponds to the cluster size or some majority fraction of the expected cluster size. For small, static clusters of 10 or fewer nodes, set this value to the number of nodes in the cluster.

### **nifi.cluster.flow.election.max.wait.time**

In an elastic cloud environment, the time to provision hosts affects the application startup time. The `nifi.cluster.flow.election.max.wait.time` property determines how long NiFi waits before deciding on a flow. Make this value commensurate with the overall launch time of the cluster at its starting size. In initial testing, five minutes are more than adequate in all Azure regions with the recommended instance types. But you can increase this value if the time to provision regularly exceeds the default.

## **Java**

We recommend using an [LTS release of Java](#). Of these releases, Java 11 is slightly preferable to Java 8 because Java 11 supports a faster garbage collection implementation. However, it's possible to have a high-performance NiFi deployment by using either release.

The following sections discuss common JVM configurations to use when running NiFi. Set JVM parameters in the bootstrap configuration file at `$NIFI_HOME/conf/bootstrap.conf`.

### **Garbage collector**

If you're running Java 11, we recommend using the G1 garbage collector (G1GC) in most situations. G1GC has improved performance over ParallelGC because G1GC reduces the length of GC pauses. G1GC is the default in Java 11, but you can configure it explicitly by setting the following value in `bootstrap.conf`:

```
java.arg.13=-XX:+UseG1GC
```

If you're running Java 8, don't use G1GC. Use ParallelGC instead. There are deficiencies in the Java 8 implementation of G1GC that prevent you from using it with the recommended repository implementations. ParallelGC is slower than G1GC. But with ParallelGC, you can still have a high-performance NiFi deployment with Java 8.

### **Heap**

A set of properties in the `bootstrap.conf` file determines the configuration of the NiFi JVM heap. For a standard flow, configure a 32-GB heap by using these settings:

```
config
```

```
java.arg.3=-Xmx32g
java.arg.2=-Xms32g
```

To choose the optimal heap size to apply to the JVM process, consider two factors:

- The characteristics of the data flow
- The way that NiFi uses memory in its processing

For detailed documentation, see [Apache NiFi in Depth](#).

Make the heap only as large as needed to fulfill the processing requirements. This approach minimizes the length of GC pauses. For general considerations for Java garbage collection, see the garbage collection tuning guide for your version of Java.

When adjusting JVM memory settings, consider these important factors:

- The number of *FlowFiles*, or NiFi data records, that are active in a given period. This number includes back-pressed or queued *FlowFiles*.
- The number of attributes that are defined in *FlowFiles*.
- The amount of memory that a processor requires to process a particular piece of content.
- The way that a processor processes data:
  - Streaming data
  - Using record-oriented processors
  - Holding all data in memory at once

These details are important. During processing, NiFi holds references and attributes for each *FlowFile* in memory. At peak performance, the amount of memory that the system uses is proportional to the number of live *FlowFiles* and all the attributes that they contain. This number includes queued *FlowFiles*. NiFi can swap to disk. But avoid this option because it hurts performance.

Also keep in mind basic object memory usage. Specifically, make your heap large enough to hold objects in memory. Consider these tips for configuring the memory settings:

- Run your flow with representative data and minimal back pressure by starting with the setting `-Xmx4G` and then increasing memory conservatively as needed.
- Run your flow with representative data and peak back pressure by starting with the setting `-Xmx4G` and then increasing cluster size conservatively as needed.
- Profile the application while the flow is running by using tools such as VisualVM and YourKit.
- If conservative increases in heap don't improve performance significantly, consider redesigning flows, processors, and other aspects of your system.

## Additional JVM parameters

The following table lists additional JVM options. It also provides the values that worked best in initial testing. Tests observed GC activity and memory usage and used careful profiling.

[Expand table](#)

Parameter	Description	JVM default	Recommendation
<code>InitiatingHeapOccupancyPercent</code>	The amount of heap that's in use before a marking cycle is triggered.	45	35
<code>ParallelGCThreads</code>	The number of threads that GC uses. This value is capped to limit the overall effect on the system.	5/8 of the number of vCPUs	8
<code>ConcGCThreads</code>	The number of GC threads to run in parallel. This value is increased to account for capped <code>ParallelGCThreads</code> .	1/4 of the <code>ParallelGCThreads</code> value	4
<code>G1ReservePercent</code>	The percentage of reserve memory to keep free. This value is increased to avoid to-space exhaustion, which helps avoid full GC.	10	20
<code>UseStringDeduplication</code>	Whether to try to identify and de-duplicate references to identical strings. Turning on this feature can result in memory savings.	-	present

Configure these settings by adding the following entries to the NiFi `bootstrap.conf`:

```
config

java.arg.17=-XX:+UseStringDeduplication
java.arg.18=-XX:G1ReservePercent=20
java.arg.19=-XX:ParallelGCThreads=8
java.arg.20=-XX:ConcGCThreads=4
java.arg.21=-XX:InitiatingHeapOccupancyPercent=35
```

## ZooKeeper

For improved fault tolerance, run ZooKeeper as a cluster. Take this approach even though most NiFi deployments put a relatively modest load on ZooKeeper. Turn on clustering for ZooKeeper explicitly. By default, ZooKeeper runs in single-server mode. For detailed information, see [Clustered \(Multi-Server\) Setup](#) in the ZooKeeper Administrator's Guide.

Except for the clustering settings, use default values for your ZooKeeper configuration.

If you have a large NiFi cluster, you might need to use a greater number of ZooKeeper servers. For smaller cluster sizes, smaller VM sizes and Standard SSD managed disks are sufficient.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Muazma Zahid](#) | Principal PM Manager

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

The material and recommendations in this document came from several sources:

- Experimentation
- Azure best practices
- NiFi community knowledge, best practices, and documentation

For more information, see the following resources:

- [Apache NiFi System Administrator's Guide](#)
- [Apache NiFi mailing lists](#)
- [Cloudera best practices for setting up a high-performance NiFi installation](#)
- [Azure premium storage: design for high performance](#)
- [Troubleshoot Azure virtual machine performance on Linux or Windows](#)

## Related resources

- [Apache NiFi monitoring with MonitoFi](#)
- [Helm-based deployments for Apache NiFi](#)
- [Azure Data Explorer monitoring](#)
- [\[Hybrid ETL with Azure Data Factory\]\[Hybrid ETL with Azure Data Factory\]](#)
- [\[DataOps for the modern data warehouse\]\[DataOps for the modern data warehouse\]](#)
- [Data warehousing and analytics](#)

# Automated enterprise BI

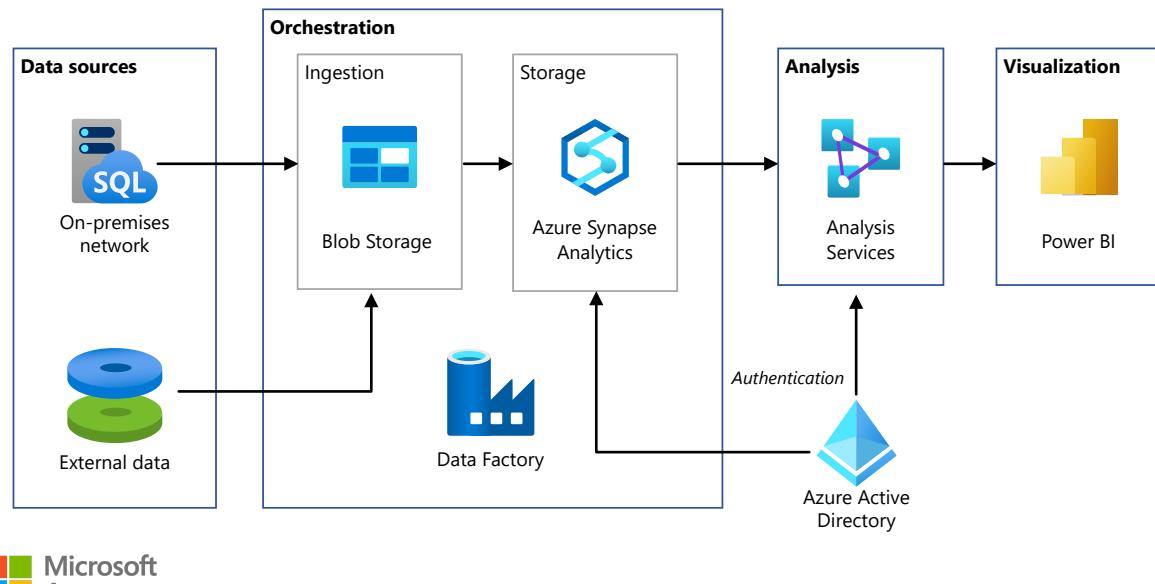
Microsoft Entra ID   Azure Analysis Services   Azure Blob Storage   Azure Data Factory  
Azure Synapse Analytics

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This example is about how to perform incremental loading in an [extract, load, and transform \(ELT\)](#) pipeline. It uses Azure Data Factory to automate the ELT pipeline. The pipeline incrementally moves the latest OLTP data from an on-premises SQL Server database into Azure Synapse. Transactional data is transformed into a tabular model for analysis.

## Architecture



Download a [Visio file](#) of this architecture.

This architecture builds on the one shown in [Enterprise BI with Azure Synapse](#), but adds some features that are important for enterprise data warehousing scenarios.

- Automation of the pipeline using Data Factory.

- Incremental loading.
- Integrating multiple data sources.
- Loading binary data such as geospatial data and images.

## Workflow

The architecture consists of the following services and components.

### Data sources

**On-premises SQL Server.** The source data is located in a SQL Server database on premises. To simulate the on-premises environment. The [Wide World Importers OLTP sample database](#) is used as the source database.

**External data.** A common scenario for data warehouses is to integrate multiple data sources. This reference architecture loads an external data set that contains city populations by year, and integrates it with the data from the OLTP database. You can use this data for insights such as: "Does sales growth in each region match or exceed population growth?"

### Ingestion and data storage

**Blob Storage.** Blob storage is used as a staging area for the source data before loading it into Azure Synapse.

**Azure Synapse.** [Azure Synapse](#) is a distributed system designed to perform analytics on large data. It supports massive parallel processing (MPP), which makes it suitable for running high-performance analytics.

**Azure Data Factory.** [Data Factory](#) is a managed service that orchestrates and automates data movement and data transformation. In this architecture, it coordinates the various stages of the ELT process.

### Analysis and reporting

**Azure Analysis Services.** [Analysis Services](#) is a fully managed service that provides data modeling capabilities. The semantic model is loaded into Analysis Services.

**Power BI.** Power BI is a suite of business analytics tools to analyze data for business insights. In this architecture, it queries the semantic model stored in Analysis Services.

## Authentication

Microsoft Entra ID (Microsoft Entra ID) authenticates users who connect to the Analysis Services server through Power BI.

Data Factory can also use Microsoft Entra ID to authenticate to Azure Synapse, by using a service principal or Managed Service Identity (MSI).

## Components

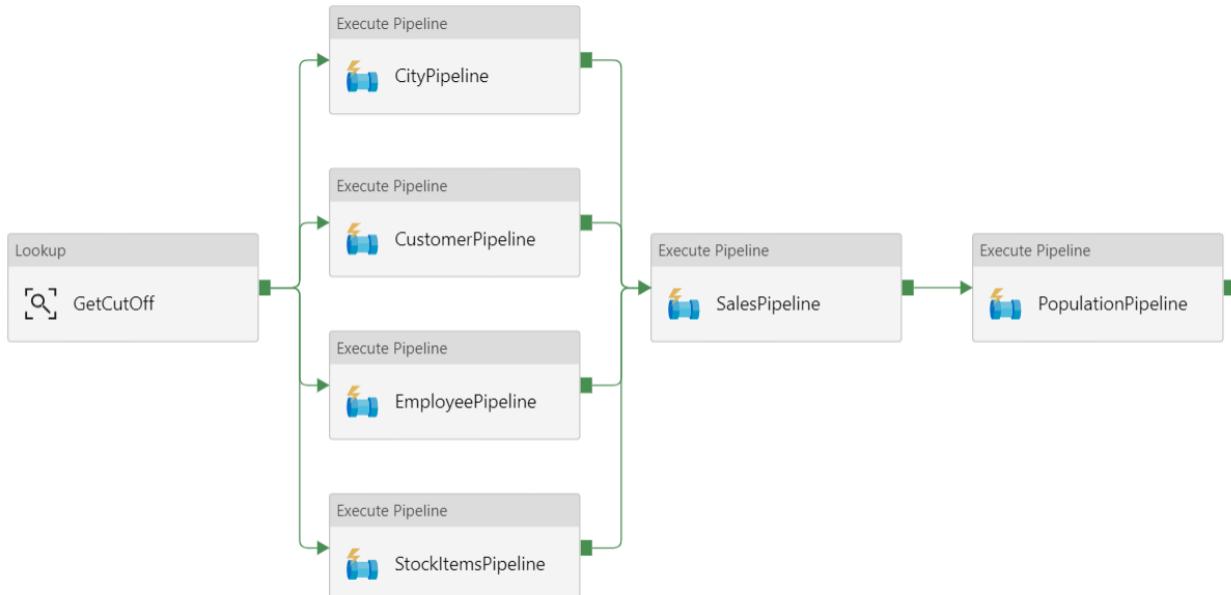
- [Azure Blob Storage](#)
- [Azure Synapse Analytics](#)
- [Azure Data Factory](#)
- [Azure Analysis Services](#)
- [Power BI](#)
- [Microsoft Entra ID](#)

## Scenario details

### Data pipeline

In [Azure Data Factory](#), a pipeline is a logical grouping of activities used to coordinate a task — in this case, loading and transforming data into Azure Synapse.

This reference architecture defines a parent pipeline that runs a sequence of child pipelines. Each child pipeline loads data into one or more data warehouse tables.



# Recommendations

## Incremental loading

When you run an automated ETL or ELT process, it's most efficient to load only the data that changed since the previous run. This is called an *incremental load*, as opposed to a full load that loads all the data. To perform an incremental load, you need a way to identify which data has changed. The most common approach is to use a *high water mark* value, which means tracking the latest value of some column in the source table, either a datetime column or a unique integer column.

Starting with SQL Server 2016, you can use [temporal tables](#). These are system-versioned tables that keep a full history of data changes. The database engine automatically records the history of every change in a separate history table. You can query the historical data by adding a FOR SYSTEM\_TIME clause to a query. Internally, the database engine queries the history table, but this is transparent to the application.

### Note

For earlier versions of SQL Server, you can use [Change Data Capture \(CDC\)](#). This approach is less convenient than temporal tables, because you have to query a separate change table, and changes are tracked by a log sequence number, rather than a timestamp.

Temporal tables are useful for dimension data, which can change over time. Fact tables usually represent an immutable transaction such as a sale, in which case keeping the system version history doesn't make sense. Instead, transactions usually have a column that represents the transaction date, which can be used as the watermark value. For example, in the Wide World Importers OLTP database, the Sales.Invoices and Sales.InvoiceLines tables have a `LastEditedWhen` field that defaults to `sysdatetime()`.

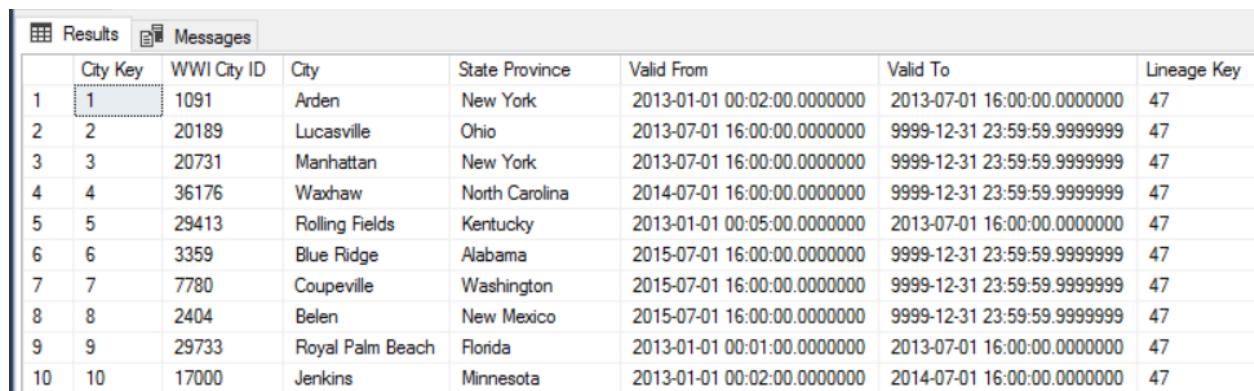
Here is the general flow for the ELT pipeline:

1. For each table in the source database, track the cutoff time when the last ELT job ran. Store this information in the data warehouse. (On initial setup, all times are set to '1-1-1900').
2. During the data export step, the cutoff time is passed as a parameter to a set of stored procedures in the source database. These stored procedures query for any records that were changed or created after the cutoff time. For the Sales fact table,

the `LastEditedWhen` column is used. For the dimension data, system-versioned temporal tables are used.

3. When the data migration is complete, update the table that stores the cutoff times.

It's also useful to record a *lineage* for each ELT run. For a given record, the lineage associates that record with the ELT run that produced the data. For each ETL run, a new lineage record is created for every table, showing the starting and ending load times. The lineage keys for each record are stored in the dimension and fact tables.



	City Key	WWI City ID	City	State Province	Valid From	Valid To	Lineage Key
1	1	1091	Arden	New York	2013-01-01 00:02:00.0000000	2013-07-01 16:00:00.0000000	47
2	2	20189	Lucasville	Ohio	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
3	3	20731	Manhattan	New York	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
4	4	36176	Waxhaw	North Carolina	2014-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
5	5	29413	Rolling Fields	Kentucky	2013-01-01 00:05:00.0000000	2013-07-01 16:00:00.0000000	47
6	6	3359	Blue Ridge	Alabama	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
7	7	7780	Coupeville	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
8	8	2404	Belen	New Mexico	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
9	9	29733	Royal Palm Beach	Florida	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000	47
10	10	17000	Jenkins	Minnesota	2013-01-01 00:02:00.0000000	2014-07-01 16:00:00.0000000	47

After a new batch of data is loaded into the warehouse, refresh the Analysis Services tabular model. See [Asynchronous refresh with the REST API](#).

## Data cleansing

Data cleansing should be part of the ELT process. In this reference architecture, one source of bad data is the city population table, where some cities have zero population, perhaps because no data was available. During processing, the ELT pipeline removes those cities from the city population table. Perform data cleansing on staging tables, rather than external tables.

## External data sources

Data warehouses often consolidate data from multiple sources. For example, an external data source that contains demographics data. This dataset is available in Azure blob storage as part of the [WorldWideImportersDW](#) sample.

Azure Data Factory can copy directly from blob storage, using the [blob storage connector](#). However, the connector requires a connection string or a shared access signature, so it can't be used to copy a blob with public read access. As a workaround, you can use PolyBase to create an external table over Blob storage and then copy the external tables into Azure Synapse.

## Handling large binary data

For example, in the source database, a City table has a Location column that holds a [geography](#) spatial data type. Azure Synapse doesn't support the [geography](#) type natively, so this field is converted to a [varbinary](#) type during loading. (See [Workarounds for unsupported data types](#).)

However, PolyBase supports a maximum column size of `varbinary(8000)`, which means some data could be truncated. A workaround for this problem is to break the data up into chunks during export, and then reassemble the chunks, as follows:

1. Create a temporary staging table for the Location column.
2. For each city, split the location data into 8000-byte chunks, resulting in 1 – N rows for each city.
3. To reassemble the chunks, use the T-SQL [PIVOT](#) operator to convert rows into columns and then concatenate the column values for each city.

The challenge is that each city will be split into a different number of rows, depending on the size of geography data. For the PIVOT operator to work, every city must have the same number of rows. To make this work, the T-SQL query does some tricks to pad out the rows with blank values, so that every city has the same number of columns after the pivot. The resulting query turns out to be much faster than looping through the rows one at a time.

The same approach is used for image data.

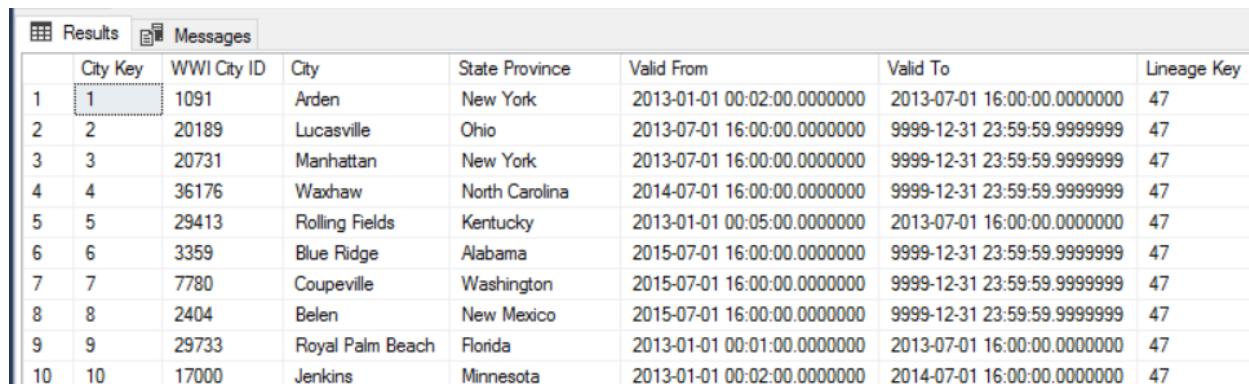
## Slowly changing dimensions

Dimension data is relatively static, but it can change. For example, a product might get reassigned to a different product category. There are several approaches to handling slowly changing dimensions. A common technique, called [Type 2](#), is to add a new record whenever a dimension changes.

In order to implement the Type 2 approach, dimension tables need additional columns that specify the effective date range for a given record. Also, primary keys from the source database will be duplicated, so the dimension table must have an artificial primary key.

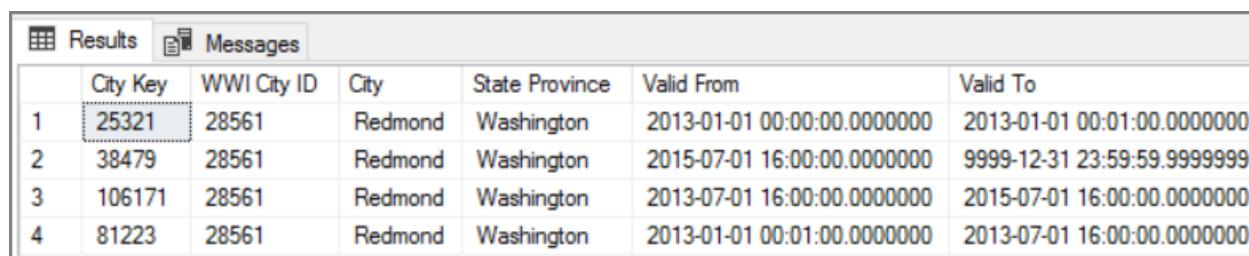
For example, the following image shows the Dimension.City table. The `WWI_City_ID` column is the primary key from the source database. The `city Key` column is an artificial key generated during the ETL pipeline. Also notice that the table has `Valid From` and

`Valid To` columns, which define the range when each row was valid. Current values have a `Valid To` equal to '9999-12-31'.



	City Key	WWI City ID	City	State Province	Valid From	Valid To	Lineage Key
1	1	1091	Arden	New York	2013-01-01 00:02:00.0000000	2013-07-01 16:00:00.0000000	47
2	2	20189	Lucasville	Ohio	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
3	3	20731	Manhattan	New York	2013-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
4	4	36176	Waxhaw	North Carolina	2014-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
5	5	29413	Rolling Fields	Kentucky	2013-01-01 00:05:00.0000000	2013-07-01 16:00:00.0000000	47
6	6	3359	Blue Ridge	Alabama	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
7	7	7780	Coupeville	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
8	8	2404	Belen	New Mexico	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999	47
9	9	29733	Royal Palm Beach	Florida	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000	47
10	10	17000	Jenkins	Minnesota	2013-01-01 00:02:00.0000000	2014-07-01 16:00:00.0000000	47

The advantage of this approach is that it preserves historical data, which can be valuable for analysis. However, it also means there will be multiple rows for the same entity. For example, here are the records that match `WWI City ID` = 28561:



	City Key	WWI City ID	City	State Province	Valid From	Valid To
1	25321	28561	Redmond	Washington	2013-01-01 00:00:00.0000000	2013-01-01 00:01:00.0000000
2	38479	28561	Redmond	Washington	2015-07-01 16:00:00.0000000	9999-12-31 23:59:59.9999999
3	106171	28561	Redmond	Washington	2013-07-01 16:00:00.0000000	2015-07-01 16:00:00.0000000
4	81223	28561	Redmond	Washington	2013-01-01 00:01:00.0000000	2013-07-01 16:00:00.0000000

For each Sales fact, you want to associate that fact with a single row in City dimension table, corresponding to the invoice date.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

For additional security, you can use [Virtual Network service endpoints](#) to secure Azure service resources to only your virtual network. This fully removes public Internet access to those resources, allowing traffic only from your virtual network.

With this approach, you create a VNet in Azure and then create private service endpoints for Azure services. Those services are then restricted to traffic from that virtual

network. You can also reach them from your on-premises network through a gateway.

Be aware of the following limitations:

- If service endpoints are enabled for Azure Storage, PolyBase cannot copy data from Storage into Azure Synapse. There is a mitigation for this issue. For more information, see [Impact of using VNet Service Endpoints with Azure storage](#).
- To move data from on-premises into Azure Storage, you will need to allow public IP addresses from your on-premises or ExpressRoute. For details, see [Securing Azure services to virtual networks](#).
- To enable Analysis Services to read data from Azure Synapse, deploy a Windows VM to the virtual network that contains the Azure Synapse service endpoint. Install [Azure On-premises Data Gateway](#) on this VM. Then connect your Azure Analysis service to the data gateway.

## DevOps

- Create separate resource groups for production, development, and test environments. Separate resource groups make it easier to manage deployments, delete test deployments, and assign access rights.
- Put each workload in a separate deployment template and store the resources in source control systems. You can deploy the templates together or individually as part of a CI/CD process, making the automation process easier.

In this architecture, there are three main workloads:

- The data warehouse server, Analysis Services, and related resources.
- Azure Data Factory.
- An on-premises to cloud simulated scenario.

Each workload has its own deployment template.

The data warehouse server is set up and configured by using Azure CLI commands which follows the imperative approach of the IaC practice. Consider using deployment scripts and integrate them in the automation process.

- Consider staging your workloads. Deploy to various stages and run validation checks at each stage before moving to the next stage. That way you can push updates to your production environments in a highly controlled way and minimize unanticipated deployment issues. Use [Blue-green deployment](#) and [Canary releases](#) strategies for updating live production environments.

Have a good rollback strategy for handling failed deployments. For example, you can automatically redeploy an earlier, successful deployment from your deployment history. See the `--rollback-on-error` flag parameter in Azure CLI.

- [Azure Monitor](#) is the recommended option for analyzing the performance of your data warehouse and the entire Azure analytics platform for an integrated monitoring experience. [Azure Synapse Analytics](#) provides a monitoring experience within the Azure portal to show insights to your data warehouse workload. The Azure portal is the recommended tool when monitoring your data warehouse because it provides configurable retention periods, alerts, recommendations, and customizable charts and dashboards for metrics and logs.

For more information, see the DevOps section in [Microsoft Azure Well-Architected Framework](#).

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Use the [Azure pricing calculator](#) to estimate costs. Here are some considerations for services used in this reference architecture.

## Azure Data Factory

Azure Data Factory automates the ELT pipeline. The pipeline moves the data from an on-premises SQL Server database into Azure Synapse. The data is then transformed into a tabular model for analysis. For this scenario, pricing starts from \$ 0.001 activity runs per month that includes activity, trigger, and debug runs. That price is the base charge only for orchestration. You are also charged for execution activities, such as copying data, lookups, and external activities. Each activity is individually priced. You are also charged for pipelines with no associated triggers or runs within the month. All activities are prorated by the minute and rounded up.

## Example cost analysis

Consider a use case where there are two lookups activities from two different sources. One takes 1 minute and 2 seconds (rounded up to 2 minutes) and the other one takes 1 minute resulting in total time of 3 minutes. One data copy activity takes 10 minutes. One

stored procedure activity takes 2 minutes. Total activity runs for 4 minutes. Cost is calculated as follows:

Activity runs:  $4 * \$0.001 = \$0.004$

Lookups:  $3 * (\$0.005 / 60) = \$0.00025$

Stored procedure:  $2 * (\$0.00025 / 60) = \$0.000008$

Data copy:  $10 * (\$0.25 / 60) * 4 \text{ data integration unit (DIU)} = \$0.167$

- Total cost per pipeline run: \$0.17.
- Run once per day for 30 days: \$5.1 month.
- Run once per day per 100 tables for 30 days: \$ 510

Every activity has an associated cost. Understand the pricing model and use the [ADF pricing calculator](#) to get a solution optimized not only for performance but also for cost. Manage your costs by starting, stopping, pausing, and scaling your services.

## Azure Synapse

Azure Synapse is ideal for intensive workloads with higher query performance and compute scalability needs. You can choose the pay-as-you-go model or use reserved plans of one year (37% savings) or 3 years (65% savings).

Data storage is charged separately. Other services such as disaster recovery and threat detection are also charged separately.

For more information, see [Azure Synapse Pricing](#).

## Analysis Services

Pricing for Azure Analysis Services depends on the tier. The reference implementation of this architecture uses the **Developer** tier, which is recommended for evaluation, development, and test scenarios. Other tiers include, the **Basic** tier, which is recommended for small production environment; the **Standard** tier for mission-critical production applications. For more information, see [The right tier when you need it](#).

No charges apply when you pause your instance.

For more information, see [Azure Analysis Services pricing](#).

## Blob Storage

Consider using the Azure Storage reserved capacity feature to lower cost on storage. With this model, you get a discount if you can commit to reservation for fixed storage capacity for one or three years. For more information, see [Optimize costs for Blob storage with reserved capacity](#).

For more information, see the Cost section in [Microsoft Azure Well-Architected Framework](#).

## Next steps

- [Introduction to Azure Synapse Analytics](#)
- [Get Started with Azure Synapse Analytics](#)
- [Introduction to Azure Data Factory](#)
- [What is Azure Data Factory?](#)
- [Azure Data Factory tutorials](#)

## Related resources

You may want to review the following [Azure example scenarios](#) that demonstrate specific solutions using some of the same technologies:

- [Data warehousing and analytics for sales and marketing](#)
- [Enterprise BI in Azure with Azure Synapse](#).

# Data governance with Profisee and Microsoft Purview

Azure Data Factory

Azure Kubernetes Service (AKS)

Azure Synapse Analytics

Microsoft Purview

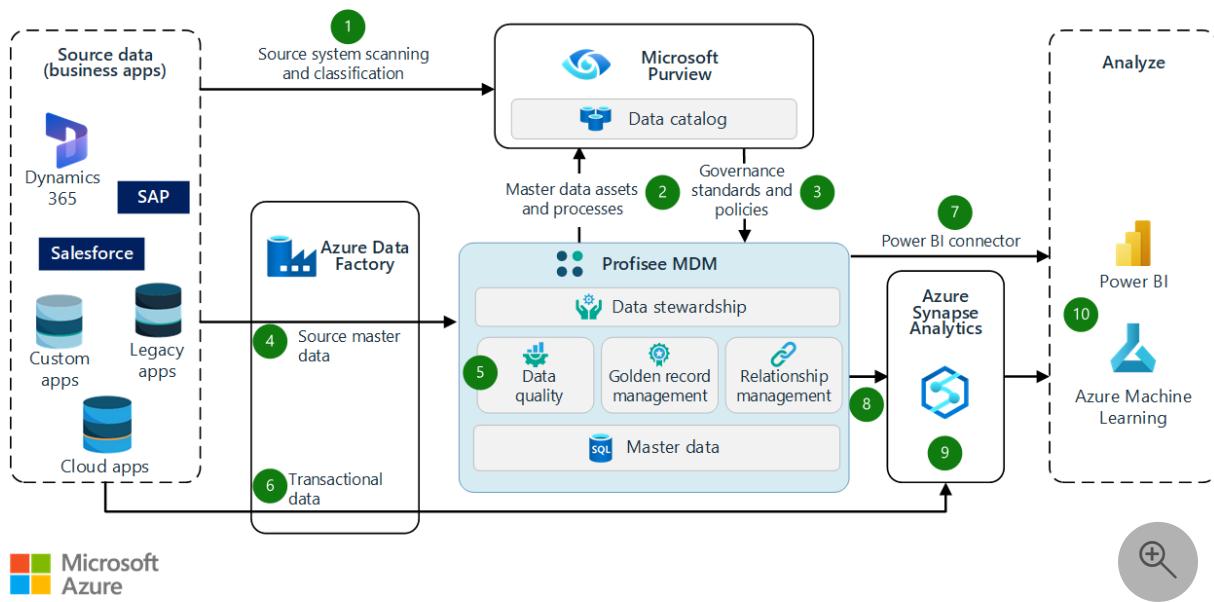
Power BI

Enterprise systems can have multiple sources of master data—the common data that's shared across systems. This fact can become apparent when you catalog data sources. Examples of master data include customer, product, location, asset, and vendor data. When you use Profisee to merge, validate, and correct your master data, you can make that data effective. Specifically, you can use it to build a common trusted platform for analytics and operational improvement. By using the governance definitions, insights, and expertise that are detailed in Microsoft Purview, you can build your platform effectively.

This reference architecture presents a governance and data management solution that features Microsoft Purview and the Profisee master data management (MDM) platform. These services work together to provide a foundation of high-quality, trusted data that maximizes the business value of data in Azure. For a short video about this solution, see [The power of fully integrated master data management in Azure](#).

## Architecture

The following diagram shows the steps that you take when you develop and operate your master data solution. Think of these steps as highly iterative. As your solution evolves, you might repeat these steps and phases, sometimes automatically and sometimes manually. Whether you use automatic or manual steps depends on the changes that your master data solution, metadata, and data undergo.



[Download a Visio file](#) of this architecture.

## Dataflow

Metadata and data flow include these steps, which are shown in the preceding figure:

1. Pre-built Microsoft Purview connectors are used to build a data catalog from source business applications. The connectors scan data sources and populate the Microsoft Purview Data Catalog.
2. The master data model is published to Microsoft Purview. Master data entities that are created in Profisee MDM are seamlessly published to Microsoft Purview. This step further populates the Microsoft Purview Data Catalog and ensures that there's a record of this critical source of data in Microsoft Purview.
3. Governance standards and policies for data stewardship are used to enrich master data entity definitions. The data is enriched in Microsoft Purview with data dictionary and glossary information, ownership data, and sensitive data classifications. Any definitions and metadata that are available in Microsoft Purview are visible in real time in Profisee as guidance for the MDM data stewards.
4. Master data from source systems is loaded into Profisee MDM. A data integration toolset like Azure Data Factory extracts data from the source systems by using any of more than 100 pre-built connectors or a REST gateway. Multiple streams of master data are loaded into Profisee MDM.
5. The master data is standardized, matched, merged, enriched, and validated according to governance rules. Other systems, like Microsoft Purview, might define data quality and governance rules. But Profisee MDM is the system that enforces

these rules. Source records are matched and merged within and across source systems to create the most complete and correct record possible. Data quality rules check each record for compliance with business and technical requirements. Any record that fails validation or that returns a low probability score is subject to remediation. To remediate failed validations, a workflow process assigns records that require review to data stewards who are experts in their business data domain. After a record has been verified or corrected, it's ready to use as a *golden record* master.

6. Transactional data is loaded into a downstream analytics solution. A data integration toolset like Data Factory extracts transactional data from source systems by using any of more than 100 pre-built connectors or a REST gateway. The toolset loads the data directly into an analytics data platform like Azure Synapse Analytics. Analysis on this raw information without the proper master golden data is subject to inaccuracy, because data overlaps, mismatches, and conflicts aren't yet resolved.
7. Power BI connectors provide direct access to the curated master data. Power BI users can use the master data directly in reports. A dedicated Power BI connector recognizes and enforces role-based security. It also hides various system fields to simplify use.
8. High-quality, curated master data is published to a downstream analytics solution. If master data records have been merged into a single golden record, parent-child links to the original records are preserved.
9. The analytics platform has a set of data that's certified in the sense that it's complete, consistent, and accurate. That data includes properly curated master data and associated transactional data. That combination forms a solid foundation of trusted data that's available for further analysis.
10. The high-quality master data is visualized and analyzed, and machine learning models are applied. The system delivers sound insights for driving the business.

## Components

- [Microsoft Purview](#) is a data governance solution that provides broad visibility into on-premises and cloud data estates. Microsoft Purview offers a combination of data discovery and classification, lineage, metadata search and discovery, and usage insights. All these features help you manage and understand data across your enterprise data landscape.

- [Profisee MDM](#) is a fast and intuitive MDM platform that integrates seamlessly with Microsoft technologies and the Azure data management ecosystem.
- [Data Factory](#) is a hybrid data integration service. You can use Data Factory to create, schedule, and orchestrate extract, transform, and load (ETL) and extract, load, and transform (ELT) workflows. Data Factory also offers more than 100 pre-built connectors and a REST gateway that you can use to extract data from source systems.
- [Azure Synapse Analytics](#) is a fast, flexible, and trusted cloud data warehouse that uses a massive parallel processing architecture. You can use Azure Synapse Analytics to scale, compute, and store data elastically and independently.
- [Power BI](#) is a suite of business analytics tools that delivers insights throughout your organization. You can use Power BI to connect to hundreds of data sources, simplify data preparation, and drive improvised analysis. You can also produce beautiful reports and then publish them for your organization to consume on the web and on mobile devices.

## Alternatives

If you don't have a dedicated MDM application, you can find some of the technical capabilities that you need to build an MDM solution in Azure:

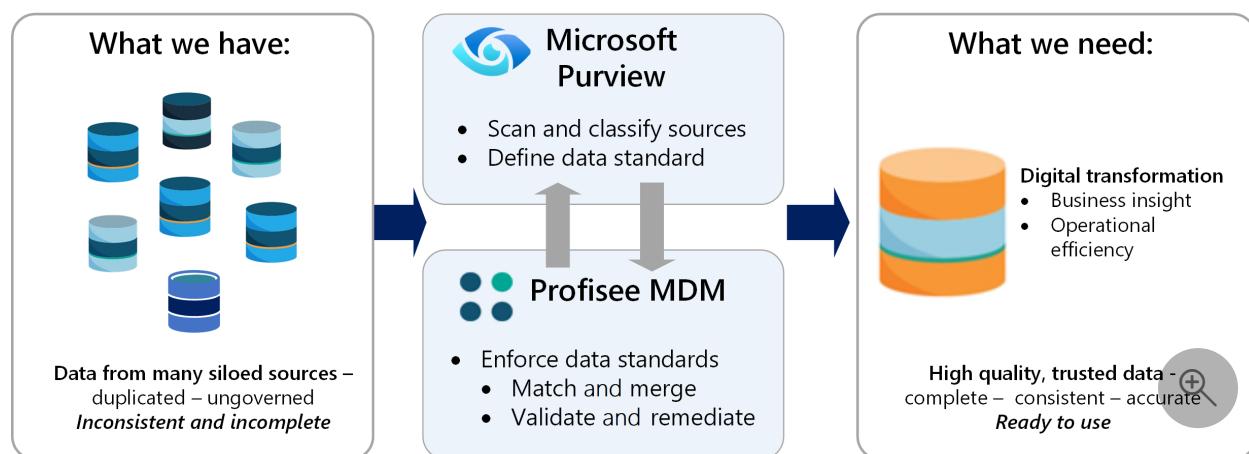
- Data quality. When you load data into an analytics platform, you can build data quality into integration processes. For example, you can use hard-coded scripts to apply data quality transformations in a [Data Factory](#) pipeline.
- Data standardization and enrichment. [Azure Maps](#) can provide data verification and standardization for address data. You can use the standardized data in Azure Functions and Data Factory. To standardize other data, you might need to develop hard-coded scripts.
- Duplicate data management. You can use Data Factory to [deduplicate rows](#) if sufficient identifiers are available for an exact match. You likely need custom hard-coded scripts to implement the logic that's needed to merge matched rows while applying appropriate data survivorship techniques.
- Data stewardship. You can use [Power Apps](#) to quickly develop basic data stewardship solutions to manage data in Azure. You can also develop appropriate user interfaces for reviews, workflows, alerts, and validations.

In Microsoft-centric environments, Azure Synapse Analytics is generally preferred as an analytics service. But you can use any analytics database. Snowflake and Databricks are common choices.

# Scenario details

As the amount of data that you load into Azure increases, the need to properly govern and manage that data across all your data sources and data consumers grows. Data that seems adequate in the source system is often found to be deficient when it's shared. It might have missing or incomplete information, or duplications and conflicts. Its overall quality might be poor. What's needed is data that's complete, consistent, and accurate.

Without high-quality data in your Azure data estate, the business value of Azure is undermined, perhaps critically. The solution is to build a foundation for data governance and management that can produce and deliver a source of truth for high-quality, trusted data. Microsoft Purview and Profisee MDM work together to form this enterprise platform.



Microsoft Purview catalogs all your data sources and identifies any sensitive information and lineage. It gives the data architect a place to consider the appropriate data standards to impose on all data. Microsoft Purview focuses on governance to find, classify, and define policies and standards. The tasks of enforcing policies and standards, cataloging data sources, and remediating deficient data fall to technologies like MDM systems.

Profisee MDM is designed to accept master data from any source. Profisee MDM then matches, merges, standardizes, verifies, corrects, and synchronizes the data across systems. This process ensures that data can be properly integrated and that it meets the needs of downstream systems, such as business intelligence (BI) and machine learning applications. The integrative Profisee platform enforces governance standards across multiple data silos.

## Better together

Microsoft Purview and Profisee MDM work better together. When integrated, they streamline data management tasks and ensure that all systems work to enforce the

same standards. Profisee MDM publishes its master data model to Microsoft Purview, where it can participate in governance. Microsoft Purview then shares the output of governance, such as a data catalog and glossary information. Profisee can review the output and enforce standards. By working jointly, Microsoft Purview and Profisee create a natural, better-together synergy that goes deeper than each independent offering.

For example, after you catalog enterprise data sources, you might determine that master data is present in multiple systems. Master data is the data that defines a domain entity. Examples of master data include customer, product, asset, location, vendor, patient, household, menu item, and ingredient data. Resolving differing definitions and matching and merging this data across systems is critical to the ability to use this data in a meaningful way. To be effective, you should merge, validate, and correct master data in Profisee MDM by using governance definitions, insights, and expertise that are detailed in Microsoft Purview. In this way, Microsoft Purview and Profisee MDM form a foundation for governance and data management, and they maximize the business value of data in Azure.

The alternative is to use whatever information you can get. But when you take this approach, you risk generating misleading results that can damage your business. When you instead use high-quality master data, you eliminate common data quality issues. Then your system delivers sound insights that you can use to drive your business, no matter which tools you use for analysis, machine learning, and visualization. Well-curated master data is a key aspect of building a solid, reliable data foundation.

When you use Profisee MDM with Microsoft Purview, you experience the following benefits:

- A common technical foundation. Profisee originated in Microsoft technologies. Profisee and Microsoft use common tools, databases, and infrastructure, which makes the Profisee solution familiar to anyone who works with Microsoft technologies. In fact, for many years, Profisee MDM was built on Microsoft Master Data Services. Now Master Data Services is nearing the end of its lifecycle, and Profisee is the premier [upgrade and replacement solution](#).
- Developer collaboration and joint development. Profisee and Microsoft Purview developers collaborate extensively to ensure a good, complementary fit between their respective solutions. This collaboration delivers a seamless integration that meets customer needs.
- Joint sales and deployments. Profisee has more MDM deployments on Azure, and jointly with Microsoft Purview, than any other MDM vendor. You can purchase Profisee through Azure Marketplace. In fiscal year 2023, Profisee is the only MDM vendor with a top-tier Microsoft partner certification that has an infrastructure as a

service (IaaS), containers as a service (CaaS), or software as a service (SaaS) offering on Azure Marketplace.

- Rapid and reliable deployment. A critical feature of all enterprise software is rapid and reliable deployment. According to the [Gartner Peer Insights](#) platform, Profisee has more implementations that take fewer than 90 days to complete than any other MDM vendor.
- Multiple domains. Profisee offers an approach to MDM that inherently uses multiple domains. There are no limitations to the number of master data domains that you can create. This design aligns well with customers who plan to modernize their data estates. A customer might start with a limited number of domains, but they ultimately benefit from maximizing their domain coverage across their whole data estate. This domain coverage is matched to their data governance coverage.
- Engineering that's designed for Azure. Profisee is engineered to be cloud native with options for SaaS and managed IaaS or CaaS deployments on Azure.

## Potential use cases

For a detailed list of MDM use cases of this solution, see [MDM use cases](#) later in this article. Key MDM use cases include the following retail and manufacturing examples:

- Consolidating customer data for analytics.
- Having a 360-degree view of product data in a consistent and accessible form, such as each product's name, description, and characteristics.
- Establishing reference data to consistently augment descriptions of master data. For example, reference data includes lists of countries/regions, currencies, colors, sizes, and units of measure.

These MDM solutions also help financial organizations that rely heavily on data for critical activities, such as timely reporting.

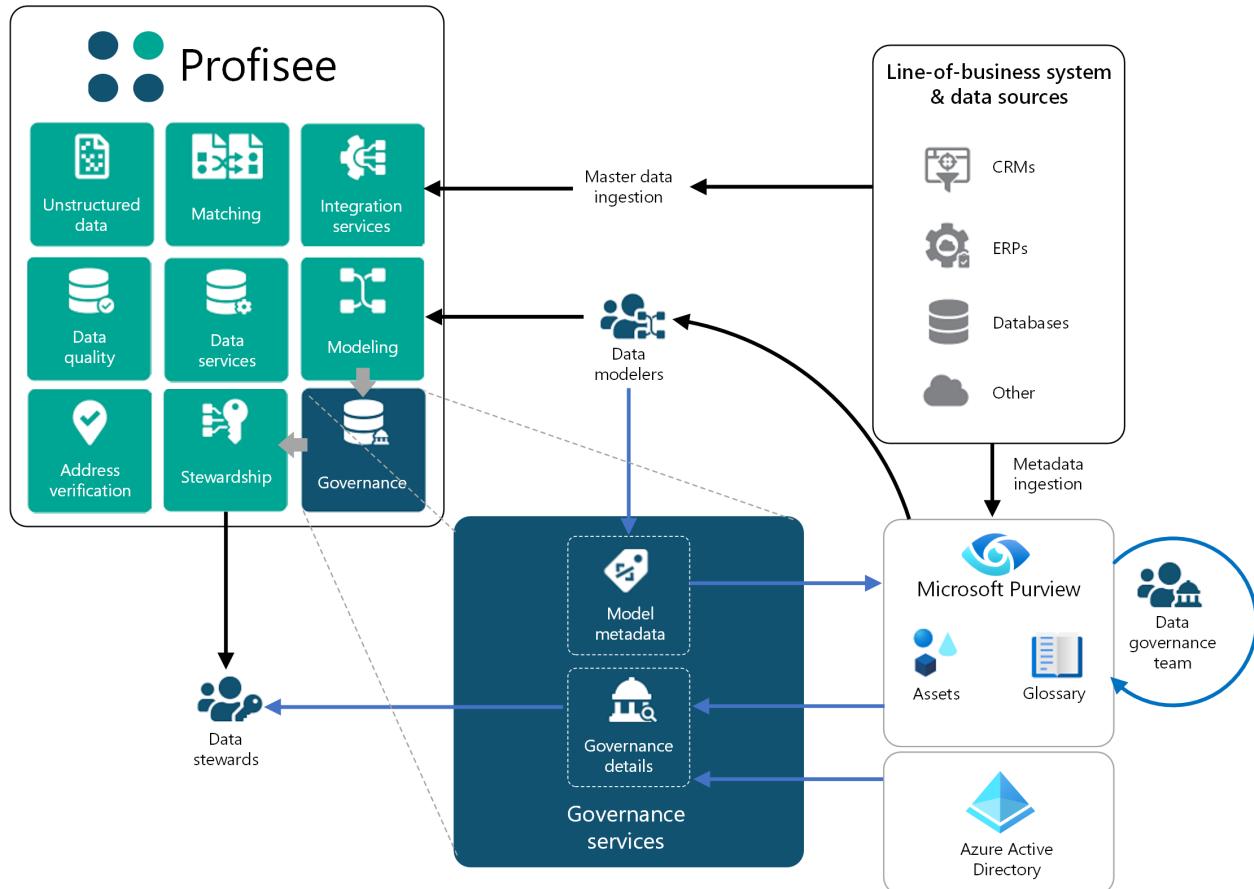
## MDM integration with Microsoft Purview

The following diagram illustrates in detail the integration of Profisee MDM in Microsoft Purview. To support this integration, the Profisee governance subsystem provides bidirectional integration with Microsoft Purview, which consists of two distinct flows:

- Solution metadata publishing occurs when your data modelers make changes to your master data model, matching strategies, and their related subartifacts. These changes are seamlessly published to Microsoft Purview as they occur. Publishing these changes syncs the metadata that's related to your master data model and

solution. As a result, the Microsoft Purview Data Catalog is further populated, and Microsoft Purview has a record of this critical data source.

- Governance details are returned and provided to data stewards and business users. These details are available as the users view data, enrich data, and remediate data quality issues by using the Profisee FastApp portal.



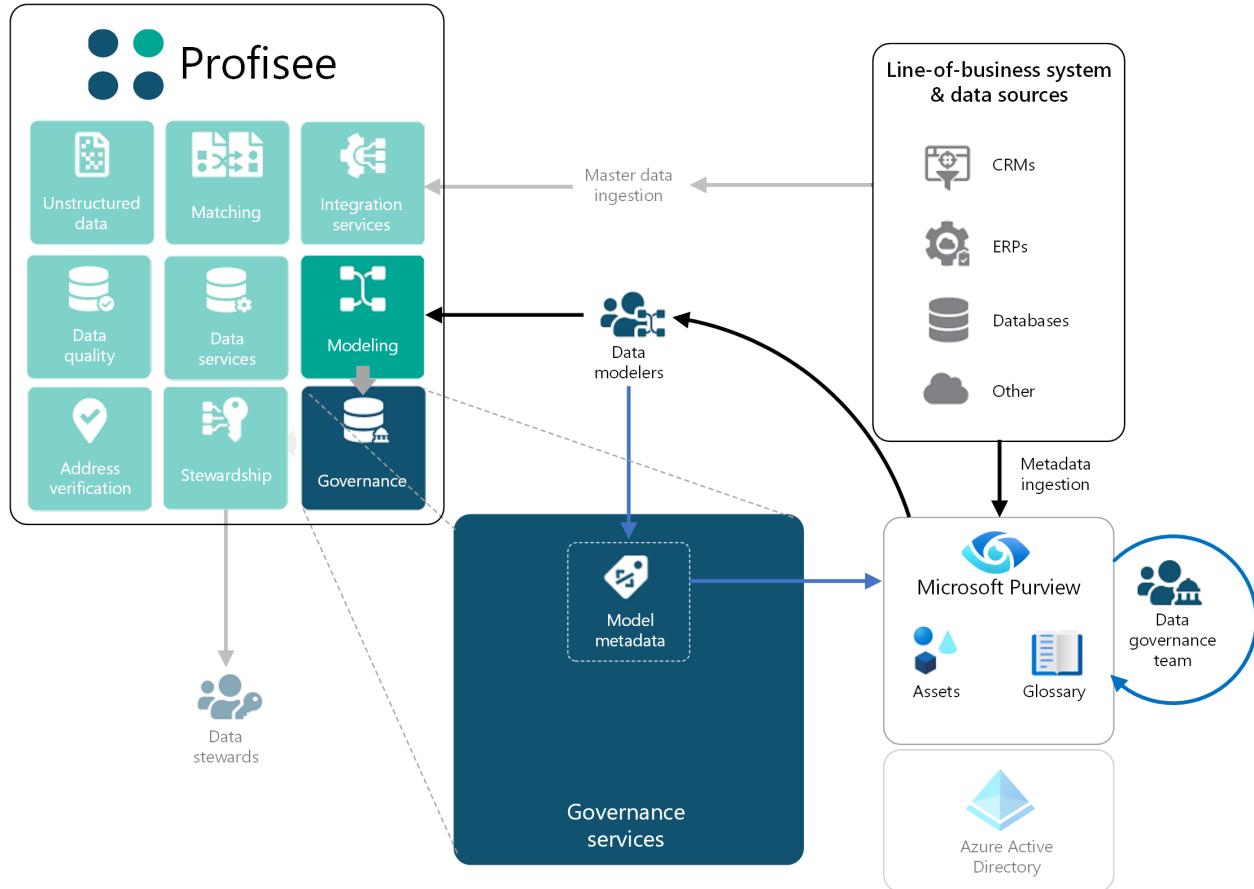
## Microsoft Purview integration capabilities

The Microsoft Purview catalog and glossary can help you maximize integration.

### Master data model design

One of the challenges of preparing an MDM solution is determining what constitutes master data and which data sources to use when you populate your master data model. You can use Microsoft Purview to help with this effort. You can take advantage of the ability to scan your critical data sources, and you can engage your data stewards and subject matter experts (SMEs). This way, you can enrich your Microsoft Purview Data Catalog with information that your stewards can then access, to better align your master data model with your line-of-business systems. You can reconcile conflicting terminology. This process yields a master data model that optimally reflects the terminology and definitions that you want to standardize for your business. It also avoids outdated and misleading verbiage.

The following excerpt from the broader diagram illustrates this integration use case. First, you use Microsoft Purview system scanning functions to ingest metadata from your line-of-business systems. Next, your data stewards and SMEs prepare a solid catalog and contacts. Then the data modelers who work with Profisee MDM modeling services can prepare and evolve your master data model. This work aligns with the standards that you define in Microsoft Purview.



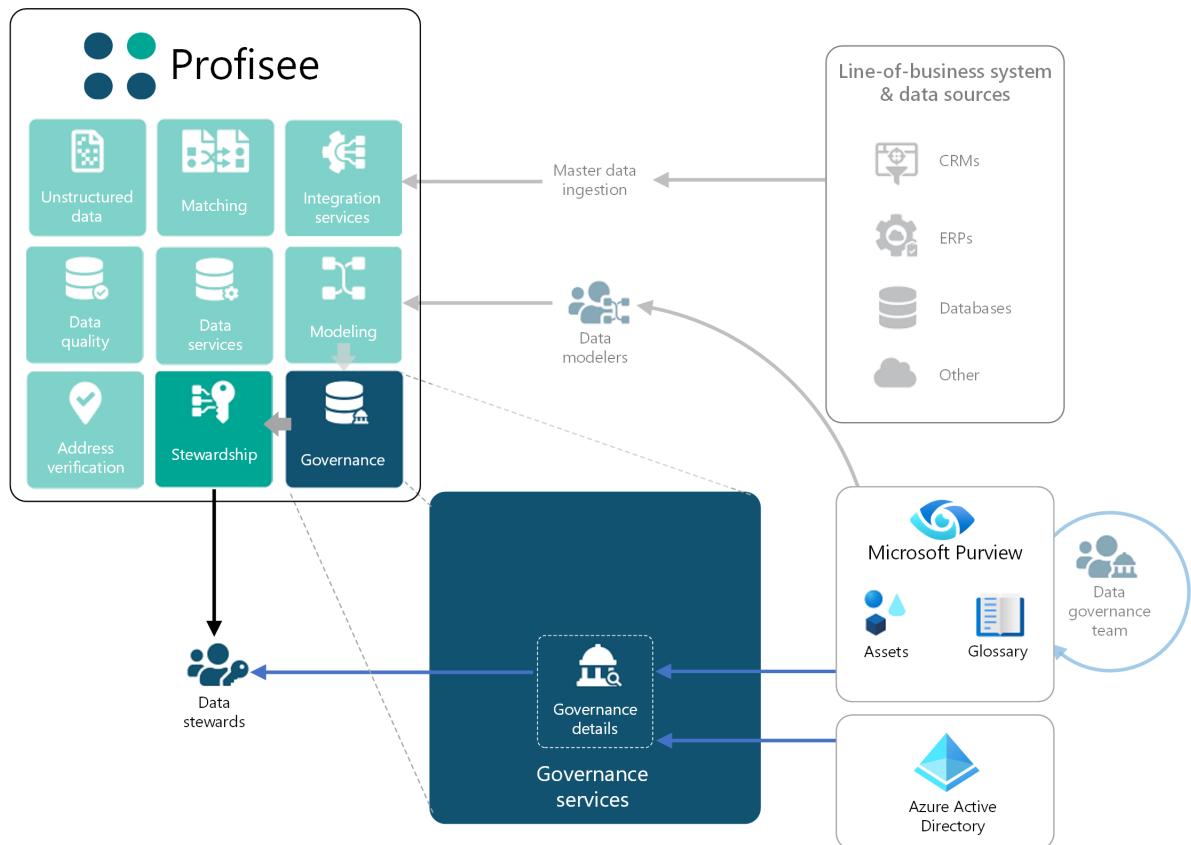
As your data stewards evolve the model, the modeling services within the Profisee MDM platform publish changes that Profisee MDM governance services receive. In turn, Profisee MDM prepares and forwards those changes to Microsoft Purview for inclusion in its updated data catalog. These additions to the catalog ensure that your master data definitions are included in the broader data estate and that they can be governed and controlled in the same manner as your line-of-business system metadata. By ensuring that this information is cataloged together, you're in a better position to manage the relationships between your master data and your line-of-business system data.

## Data stewardship

Large enterprises that have correspondingly complex and expansive data estates can present challenges to data stewards, who are responsible for managing and remediating issues as they arise. Key data domains can be complex, with many obscure attributes that only tenured employees who have significant institutional knowledge understand.

Through the Profisee MDM integration with Microsoft Purview, you can capture this institutional knowledge within Microsoft Purview and make it available for use within Profisee MDM. As a result, you alleviate a great need for corporate data knowledge when you manage critical and time-sensitive information.

The following figure illustrates the flow of information from Microsoft Purview to the data stewards who work in the Profisee FastApp portal. The governance data service integrates with Microsoft Purview and Microsoft Entra ID. This service provides lookup functionality. FastApp portal users can use this functionality to retrieve enriched governance data about the entities and the attributes that they work with.



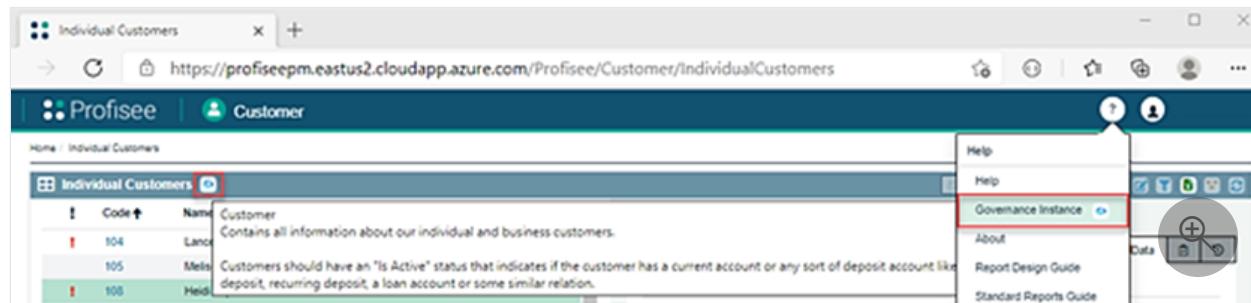
Governance services also resolve contacts that are received from Microsoft Purview to their full profile details, which are available in Microsoft Entra ID. With complete profile details, stewards can effectively collaborate with data owners and experts as they work to enhance the quality of your master data.

The Profisee MDM Governance dialog is the interface through which data stewards and users interact with governance-level details. This UI renders information that's obtained from Microsoft Purview to users. By using this information, users can review the details behind the data from which the dialog was launched. If the information that's provided in the Governance dialog is insufficient, users can go directly to the Microsoft Purview user experience.

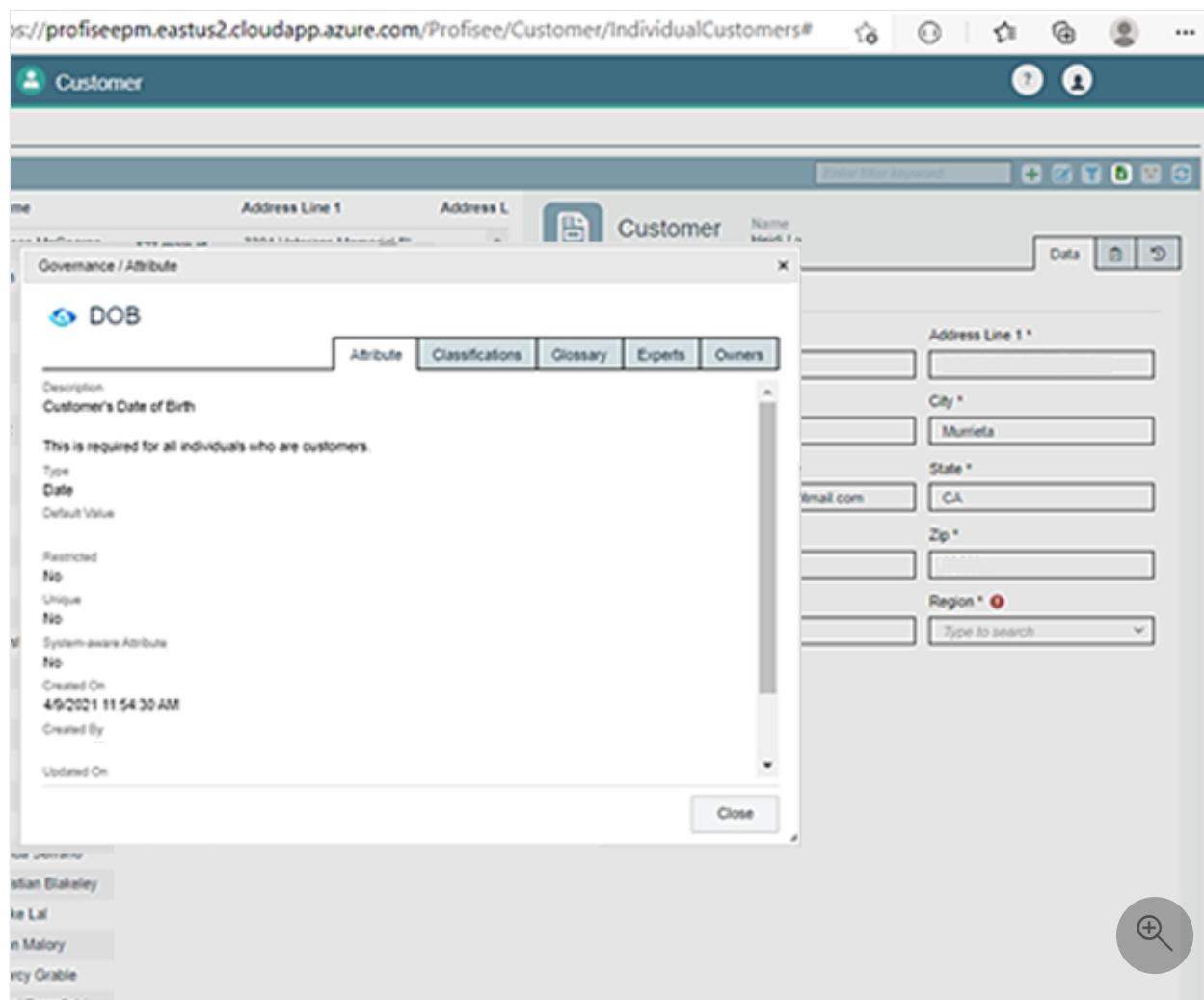
Data stewards and business users can access three Profisee MDM data asset types via the FastApp portal:

- Profisee Instance, which provides the infrastructure properties of the specific instance of the Profisee MDM platform that the user is viewing
- Profisee Entity, which provides the properties of the master data entity (the table) that the steward or user is currently viewing
- Profisee Attribute, which provides the properties of the attribute (such as the field or column) in which the user is interested

The following figure illustrates where users who are working in the FastApp portal can view governance details for each of these asset types. You can find instance-level details on the **Help** menu. You can access entity details from the page zone header, which contains an entity grid. For attribute details, go to the form that's associated with the entity grid. Access the details from the labels that are associated with the attribute.



To see summary information, hover over the governance icon, such as Microsoft Purview. Select the icon to display the full governance dialog:



To go to the full Microsoft Purview user experience, select the governance icon in the dialog header. Selecting the icon takes you to Microsoft Purview in the context of the asset that you're currently viewing. You then can easily move around in Microsoft Purview based on your discovery needs.

## MDM processing

The power of an MDM solution is in the details.

## Data modeling

The heart of your MDM solution is the underlying data model. It represents the definition of *master data* within your company. Developing a master data model involves the following tasks:

- Identify elements of source data from across your systems landscape that are critical to your company's operations and central to analyzing performance.
- Enrich the model with elements that you obtain from other third-party sources that render the data more useful, accurate, and trustworthy.

- Establish clear ownership and permissions related to the elements of your data model. This practice helps ensure that you factor visibility and change management into your model's design.

Data governance provides a critical foundation of support:

- Your governance data catalog, dictionary, glossary, and supporting resources are invaluable sources of information to your governance data stewards. These resources help stewards determine what to include in your master data model. They also help determine ownership and sensitive data classifications in Microsoft Purview. You can reinforce terminology in your model. Through this practice, you can establish an official lexicon for your business. By integrating terminology, your master data model can also translate any esoteric terms that are in use in various source systems to the approved language of the business.
- Third-party systems are often a source of master data that's separate and apart from your line-of-business systems. It's critical to add elements to your model to capture the information that these systems add to your data, and to reflect these sources of information back into your data catalog.
- You can use ownership and data access, as identified in your governance catalog, to enforce access and change management permissions within your MDM solution. As a result, you align your corporate policies and needs with the tools that you use to manage and steward your master data.

## Source data load

Ideally, your disparate line-of-business systems load data into your master data model with little or no change or transformation. The goal is to have a centralized version of the data as it exists in the source system. There should be as little loss of fidelity as possible between the source system and your master data repository. By limiting the complexity of your loading process, you make lineage simpler. And when you use technology like Data Factory pipelines, your governance solution can inspect the flow. Then your solution can identify the relationships between your source system and your master data model. Specifically, your solution can extract data from source systems by using any of more than 100 pre-built connectors and a REST gateway.

## Data enrichment and standardization

After you load source data into your model, you can extend it by tapping into rich sources of third-party data. You can use these systems to improve the data that you obtain from your line-of-business systems. You can also use these systems to augment

the source data with information that enhances its use for other downstream consumers. For example:

- You can use address-verification services like Bing to correct and improve source system addresses. These services can standardize and add missing information that's crucial to geo-location and mail delivery.
- Third-party information services like Dun & Bradstreet can provide general-purpose or industry-specific data. You can use this data to extend the value of your golden master record. Specifically, you might add information that was unavailable or in conflict in your disparate line-of-business systems.

Profisee's publish/subscribe infrastructure makes it easy to integrate your own third-party sources into your solution as needed.

The ability to understand the sources and meaning behind this data is as critical for third-party data as it is for your internal line-of-business systems. By integrating your master data model into your governance data catalog, you can manage the relationships between internal and external sources of data while enriching your model with governance details.

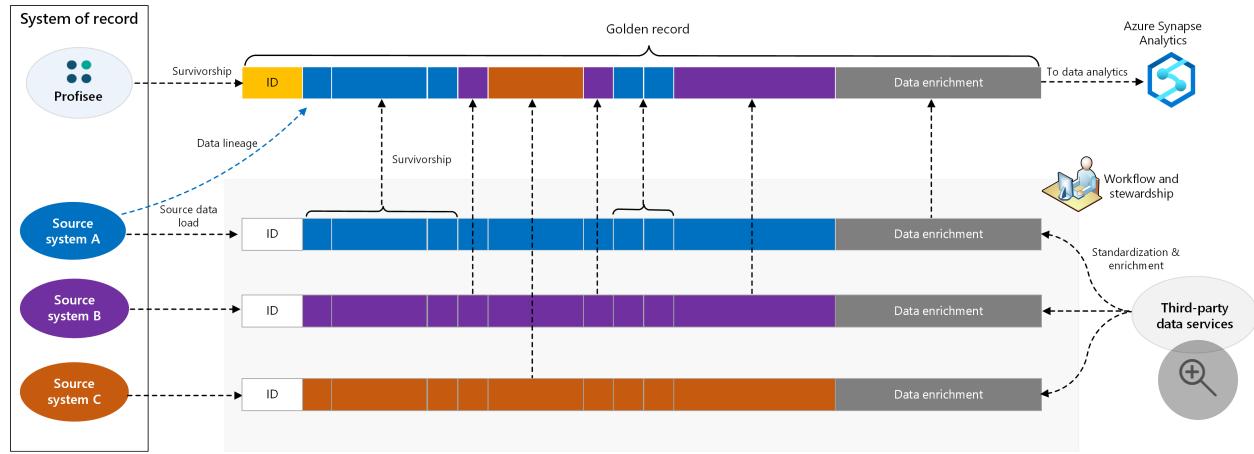
## **Data quality validation and stewardship**

After you load and enrich your data, it's important to check it for quality and adherence to standards that you establish through your governance processes. Microsoft Purview can again be a rich source of standards information. You can use Microsoft Purview to drive the data quality rules that your MDM solution enforces. Profisee MDM can also publish data quality rules as assets to your governance catalog. The rules can be subject to review and approval, which helps you provide top-down oversight of quality standards that are associated with your master data. Your rules are tied to master data entities and attributes, and those attributes can be traced back to the source system. For these reasons, you can establish the root cause of the poor data quality that originates from your line-of-business systems.

Data stewards are experts in their business domain. As stewards address issues that your master data solution reveals, they can use the Microsoft Purview data governance catalog. The catalog helps stewards understand and resolve quality issues as they arise. Backed by the support of data owners and experts, the stewards are prepared to address data quality issues quickly and accurately.

## **Matching and survivorship**

With enriched, high-quality source data, you're positioned to produce a golden record master that represents the most accurate information across your disparate line-of-business systems. The following figure illustrates how all the steps culminate in high-quality data that's ready to use for business analysis. At any time, you can sync this data across your data estate.



The Profisee MDM matching engine produces a golden record master as part of the survivorship process. Survivorship rules selectively populate the golden record with information that you've chosen across all your source systems.

The Profisee MDM history and audit tracking subsystem tracks changes that users make. This subsystem also tracks changes that system processes like survivorship make. Matching and survivorship make it possible to trace the flow of information from your source records to the master. Profisee MDM has a record of the source system that's responsible for a specific source record. You also know how disparate source records populate the golden record. As a result, you can achieve data lineage from your analytics back to the source data that your reports reference.

## MDM use cases

Although there are numerous use cases for MDM, a few use cases cover most real-world MDM implementations. These use cases focus on a single domain, but they're unlikely to be built from only that domain. Even these focused use cases most likely involve multiple domains. In each use case, MDM meets the goal of providing a 360-degree, or unified, view of essential data types.

### Customer data

Consolidating and standardizing customer data for BI analytics is the most common MDM use case. Organizations capture customer data across an increasing number of systems and applications. Duplicate customer data records result. These duplicates are

located in and across applications, and they contain inconsistencies and discrepancies. The poor quality of the customer data limits the value of modern analytics solutions. Symptoms include the following challenges:

- It's hard to answer basic business questions like, "Who are our top customers?" and "How many new customers do we have?" Answering these questions requires significant manual effort.
- You have missing and inaccurate customer information, which makes it difficult to roll up or drill down into the data.
- You're unable to uniquely identify or verify a customer across organizational and system boundaries. As a result, you're unable to analyze your customer data across systems or business units.
- You have poor-quality insights from AI and machine learning due to the poor-quality input data.

## Product data

Product data is often spread across multiple enterprise applications, such as enterprise resource planning (ERP), product lifecycle management (PLM), or e-commerce applications. As a result, it's challenging to understand the total catalog of products that have inconsistent definitions for properties, such as the product name, description, and characteristics. Different definitions of reference data complicate this situation.

Symptoms include the following challenges:

- You're unable to support different alternative hierarchical roll-up and drill-down paths for product analytics.
- With finished goods or material inventory, you have difficulty evaluating product inventory and established vendors. You also have duplicate products, which leads to excess inventory.
- It's hard to rationalize products due to conflicting definitions. This situation leads to missing or inaccurate information in analytics.

## Reference data

In the context of analytics, reference data exists as numerous lists of data. These lists are often used to further describe other sets of master data. For example, reference data includes lists of countries/regions, currencies, colors, sizes, and units of measure.

Inconsistent reference data leads to obvious errors in downstream analytics. Symptoms are:

- Multiple representations of the same value. For example, the state of Georgia is listed as *GA* and *Georgia*, which makes it difficult to consistently aggregate and drill down into data.
- Difficulty streamlining data across systems due to an inability to *crosswalk*, or map, reference data values between systems. For example, the color red is represented by *R* in the ERP system and *Red* in the PLM system.
- Difficulty tying numbers across organizations due to differences in established reference data values that are used for data categorization.

## Financial data

Financial organizations rely heavily on data for critical activities, such as monthly, quarterly, and annual reporting. Organizations that have multiple finance and accounting systems often have financial data across multiple general ledgers that needs to be consolidated to produce financial reports. MDM can provide a centralized hub to map and manage accounts, cost centers, business entities, and other financial datasets. Through the centralized hub, MDM provides a consolidated view of these datasets. Symptoms include the following challenges:

- Difficulty aggregating financial data across multiple systems into a consolidated view
- Lack of process for adding and mapping new data elements in financial systems
- Delays in producing end-of-period financial reports

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Consider these factors when you choose a data management solution for your organization.

## Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

Profisee runs natively on Azure Kubernetes Service (AKS) and Azure SQL Database. Both services offer out-of-the-box capabilities to support high availability.

# Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Profisee authenticates users by using OpenID Connect, which implements an OAuth 2.0 authentication flow. Most organizations configure Profisee MDM to authenticate users against Microsoft Entra ID, which ensures that you can apply and enforce your enterprise policies for authentication.

# Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Running costs consist of a software license and Azure consumption. For more information, contact [Profisee](#).

# Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

Profisee MDM runs natively on AKS and SQL Database. You can configure AKS to scale Profisee MDM up, down, and across your business functions. You can deploy SQL Database in numerous configurations to balance performance, scalability, and costs.

Dynamic scaling is inherent in the cloud-native architecture of Profisee, which uses microservices and containers. If you run Profisee in your cloud tenant via Kubernetes, you can dynamically scale up and out based on your load. With the [Profisee SaaS service](#) that runs on AKS, you can configure large node pools for your pods. These pools scale dynamically based on the load on the system across the multitenant infrastructure.

For detailed information about how to deploy Profisee and Microsoft Purview on AKS, see [Microsoft Purview - Profisee MDM integration](#).

# Deploy this scenario

Profisee MDM is a packaged Kubernetes service. You can deploy Profisee MDM as a [PaaS in your Azure tenant](#), in any other cloud tenant, or on-premises. You can also deploy Profisee MDM as a [SaaS that Profisee hosts and manages](#).

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributor.*

Principal author:

- [Gaurav Malhotra](#) | Principal Group PM Manager

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- Understand the capabilities of the [REST copy connector](#) in Data Factory.
- Learn more about [Profisee running natively in Azure](#).
- Learn how to deploy Profisee to Azure by using an [Azure Resource Manager template \(ARM template\)](#).
- View [Profisee Data Factory templates](#).

## Related resources

Architecture guides:

- [Extract, transform, and load \(ETL\)](#)
- [Choose a batch processing technology in Azure](#)

Reference architectures:

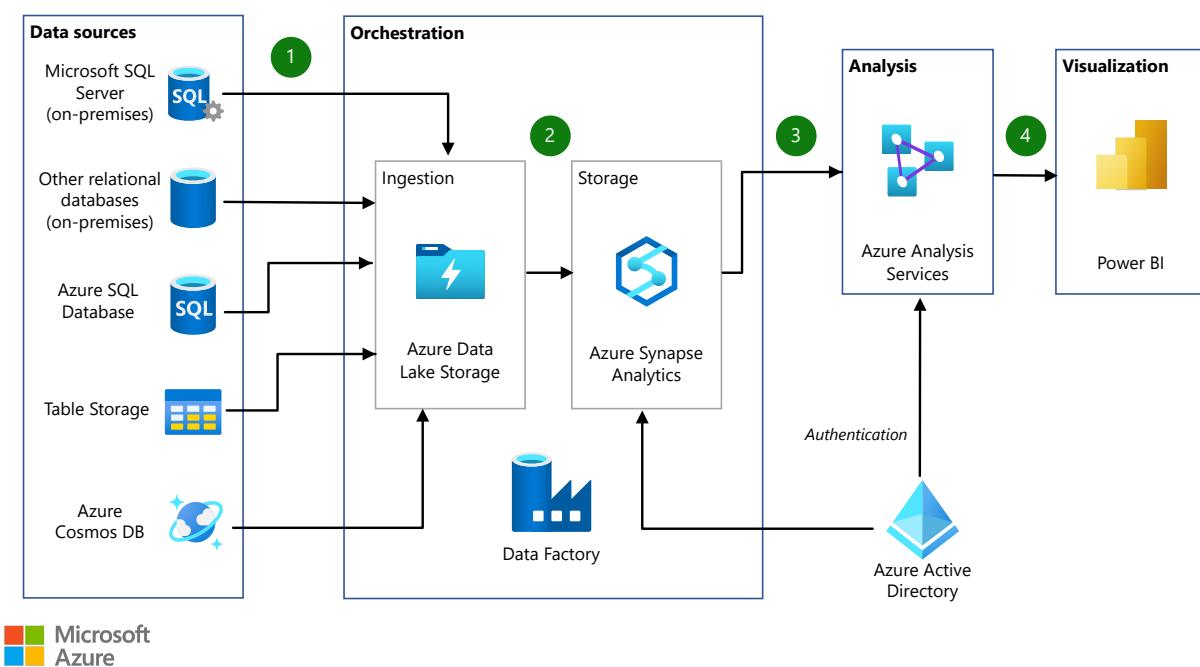
- [Master data management with Profisee and Azure Data Factory](#)
- [Analytics end-to-end with Azure Synapse](#)
- [Modern analytics architecture with Azure Databricks](#)
- [Big data analytics with enterprise-grade security using Azure Synapse](#)
- [Automated enterprise BI](#)
- [Optimize marketing with machine learning](#)
- [Enterprise business intelligence](#)

# Data warehousing and analytics

Azure Data Lake Storage   Azure Cosmos DB   Azure Data Factory   Azure SQL Database   Azure Table Storage

This example scenario demonstrates a data pipeline that integrates large amounts of data from multiple sources into a unified analytics platform in Azure. This specific scenario is based on a sales and marketing solution, but the design patterns are relevant for many industries requiring advanced analytics of large datasets such as e-commerce, retail, and healthcare.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

The data flows through the solution as follows:

1. For each data source, any updates are exported periodically into a staging area in Azure Data Lake Storage.
2. Azure Data Factory incrementally loads the data from Azure Data Lake Storage into staging tables in Azure Synapse Analytics. The data is cleansed and transformed during this process. PolyBase can parallelize the process for large datasets.
3. After loading a new batch of data into the warehouse, a previously created Azure Analysis Services tabular model is refreshed. This semantic model simplifies the

analysis of business data and relationships.

4. Business analysts use Microsoft Power BI to analyze warehoused data via the Analysis Services semantic model.

## Components

The company has data sources on many different platforms:

- SQL Server on-premises
- Oracle on-premises
- Azure SQL Database
- Azure table storage
- Azure Cosmos DB

Data is loaded from these different data sources using several Azure components:

- [Azure Data Lake Storage](#) is used to stage source data before it's loaded into Azure Synapse.
- [Data Factory](#) orchestrates the transformation of staged data into a common structure in Azure Synapse. Data Factory [uses PolyBase when loading data into Azure Synapse](#) to maximize throughput.
- [Azure Synapse](#) is a distributed system for storing and analyzing large datasets. Its use of massive parallel processing (MPP) makes it suitable for running high-performance analytics. Azure Synapse can use [PolyBase](#) to rapidly load data from Azure Data Lake Storage.
- [Analysis Services](#) provides a semantic model for your data. It can also increase system performance when analyzing your data.
- [Power BI](#) is a suite of business analytics tools to analyze data and share insights. Power BI can query a semantic model stored in Analysis Services, or it can query Azure Synapse directly.
- [Microsoft Entra ID](#) authenticates users who connect to the Analysis Services server through Power BI. Data Factory can also use Microsoft Entra ID to authenticate to Azure Synapse via a service principal or [Managed identity for Azure resources](#).

## Alternatives

- The example pipeline includes several different kinds of data sources. This architecture can handle a wide variety of relational and non-relational data sources.

- Data Factory orchestrates the workflows for your data pipeline. If you want to load data only one time or on demand, you could use tools like SQL Server bulk copy (bcp) and AzCopy to copy data into Azure Data Lake Storage. You can then load the data directly into Azure Synapse using PolyBase.
- If you have very large datasets, consider using [Data Lake Storage](#), which provides limitless storage for analytics data.
- Azure Synapse is not a good fit for OLTP workloads or data sets smaller than 250 GB. For those cases you should use Azure SQL Database or SQL Server.
- For comparisons of other alternatives, see:
  - [Choosing a data pipeline orchestration technology in Azure](#)
  - [Choosing a batch processing technology in Azure](#)
  - [Choosing an analytical data store in Azure](#)
  - [Choosing a data analytics technology in Azure](#)

## Scenario details

This example demonstrates a sales and marketing company that creates incentive programs. These programs reward customers, suppliers, salespeople, and employees. Data is fundamental to these programs, and the company wants to improve the insights gained through data analytics using Azure.

The company needs a modern approach to analysis data, so that decisions are made using the right data at the right time. The company's goals include:

- Combining different kinds of data sources into a cloud-scale platform.
- Transforming source data into a common taxonomy and structure, to make the data consistent and easily compared.
- Loading data using a highly parallelized approach that can support thousands of incentive programs, without the high costs of deploying and maintaining on-premises infrastructure.
- Greatly reducing the time needed to gather and transform data, so you can focus on analyzing the data.

## Potential use cases

This approach can also be used to:

- Establish a data warehouse to be a single source of truth for your data.
- Integrate relational data sources with other unstructured datasets.

- Use semantic modeling and powerful visualization tools for simpler data analysis.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

The technologies in this architecture were chosen because they met the company's requirements for scalability and availability, while helping them control costs.

- The [massively parallel processing architecture](#) of Azure Synapse provides scalability and high performance.
- Azure Synapse has [guaranteed SLAs](#) and [recommended practices for achieving high availability](#).
- When analysis activity is low, the company can [scale Azure Synapse on demand](#), reducing or even pausing compute to lower costs.
- Azure Analysis Services can be [scaled out](#) to reduce response times during high query workloads. You can also separate processing from the query pool, so that client queries aren't slowed down by processing operations.
- Azure Analysis Services also has [guaranteed SLAs](#) and [recommended practices for achieving high availability](#).
- The [Azure Synapse security model](#) provides connection security, [authentication and authorization](#) via Microsoft Entra ID or SQL Server authentication, and encryption. [Azure Analysis Services](#) uses Microsoft Entra ID for identity management and user authentication.

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Review a [pricing sample for a data warehousing scenario](#) via the Azure pricing calculator. Adjust the values to see how your requirements affect your costs.

- [Azure Synapse](#) allows you to scale your compute and storage levels independently. Compute resources are charged per hour, and you can scale or pause these resources on demand. Storage resources are billed per terabyte, so your costs will increase as you ingest more data.
- [Data Factory](#) costs are based on the number of read/write operations, monitoring operations, and orchestration activities performed in a workload. Your

Data Factory costs will increase with each additional data stream and the amount of data processed by each one.

- [Analysis Services](#) is available in developer, basic, and standard tiers. Instances are priced based on query processing units (QPs) and available memory. To keep your costs lower, minimize the number of queries you run, how much data they process, and how often they run.
- [Power BI](#) has different product options for different requirements. [Power BI Embedded](#) provides an Azure-based option for embedding Power BI functionality inside your applications. A Power BI Embedded instance is included in the pricing sample above.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributor.*

Principal author:

- [Alex Buck](#) | Senior Content Developer

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- Review the [Azure reference architecture for automated enterprise BI](#), which includes instructions for deploying an instance of this architecture in Azure.
- Learn more about the services used in this scenario:
  - [Introduction to Azure Data Lake Storage Gen2](#)
  - [Azure Data Factory documentation](#)
  - [What is dedicated SQL pool in Azure Synapse Analytics?](#)
  - [Azure Analysis Services documentation](#)
  - [Power BI documentation](#)
  - [Microsoft Entra documentation](#)

## Related resources

- [Enterprise data warehouse](#)

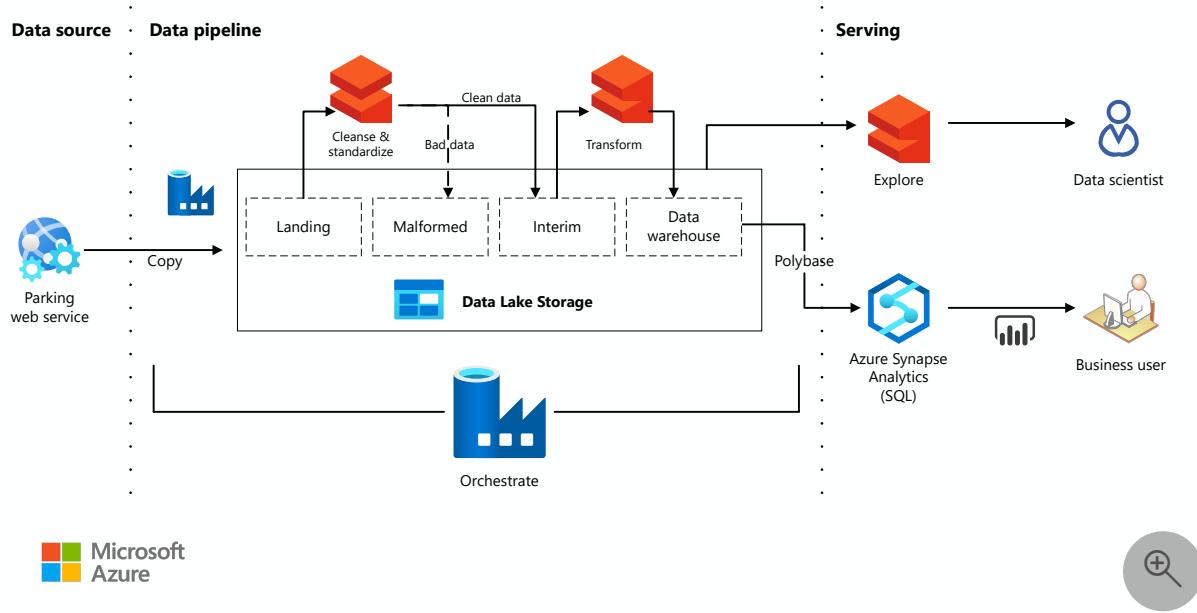
# DataOps for the modern data warehouse

Azure Data Factory   Azure Databricks   Azure DevOps   Azure Key Vault   Azure Synapse Analytics

This article describes how a fictional city planning office could use this solution. The solution provides an end-to-end data pipeline that follows the MDW architectural pattern, along with corresponding DevOps and DataOps processes, to assess parking use and make more informed business decisions.

## Architecture

The following diagram shows the overall architecture of the solution.



Download a [Visio file](#) of this architecture.

## Dataflow

Azure Data Factory (ADF) orchestrates and Azure Data Lake Storage (ADLS) Gen2 stores the data:

1. The Contoso city parking web service API is available to transfer data from the parking spots.
2. There's an ADF copy job that transfers the data into the Landing schema.

3. Next, Azure Databricks cleanses and standardizes the data. It takes the raw data and conditions it so data scientists can use it.
4. If validation reveals any bad data, it gets dumped into the Malformed schema.

**ⓘ Important**

People have asked why the data isn't validated before it's stored in ADLS. The reason is that the validation might introduce a bug that could corrupt the dataset. If you introduce a bug at this step, you can fix the bug and replay your pipeline. If you dumped the bad data before you added it to ADLS, then the corrupted data is useless because you can't replay your pipeline.

5. There's a second Azure Databricks transform step that converts the data into a format that you can store in the data warehouse.
6. Finally, the pipeline serves the data in two different ways:
  - a. Databricks makes the data available to the data scientist so they can train models.
  - b. Polybase moves the data from the data lake to Azure Synapse Analytics and Power BI accesses the data and presents it to the business user.

## Components

The solution uses these components:

- [Azure Data Factory \(ADF\)](#)
- [Azure Databricks](#)
- [Azure Data Lake Storage \(ADLS\) Gen2](#)
- [Azure Synapse Analytics](#)
- [Azure Key Vault](#)
- [Azure DevOps](#)
- [Power BI](#)

## Scenario details

A modern data warehouse (MDW) lets you easily bring all of your data together at any scale. It doesn't matter if it's structured, unstructured, or semi-structured data. You can gain insights to an MDW through analytical dashboards, operational reports, or advanced analytics for all your users.

Setting up an MDW environment for both development (dev) and production (prod) environments is complex. Automating the process is key. It helps increase productivity while minimizing the risk of errors.

This article describes how a fictional city planning office could use this solution. The solution provides an end-to-end data pipeline that follows the MDW architectural pattern, along with corresponding DevOps and DataOps processes, to assess parking use and make more informed business decisions.

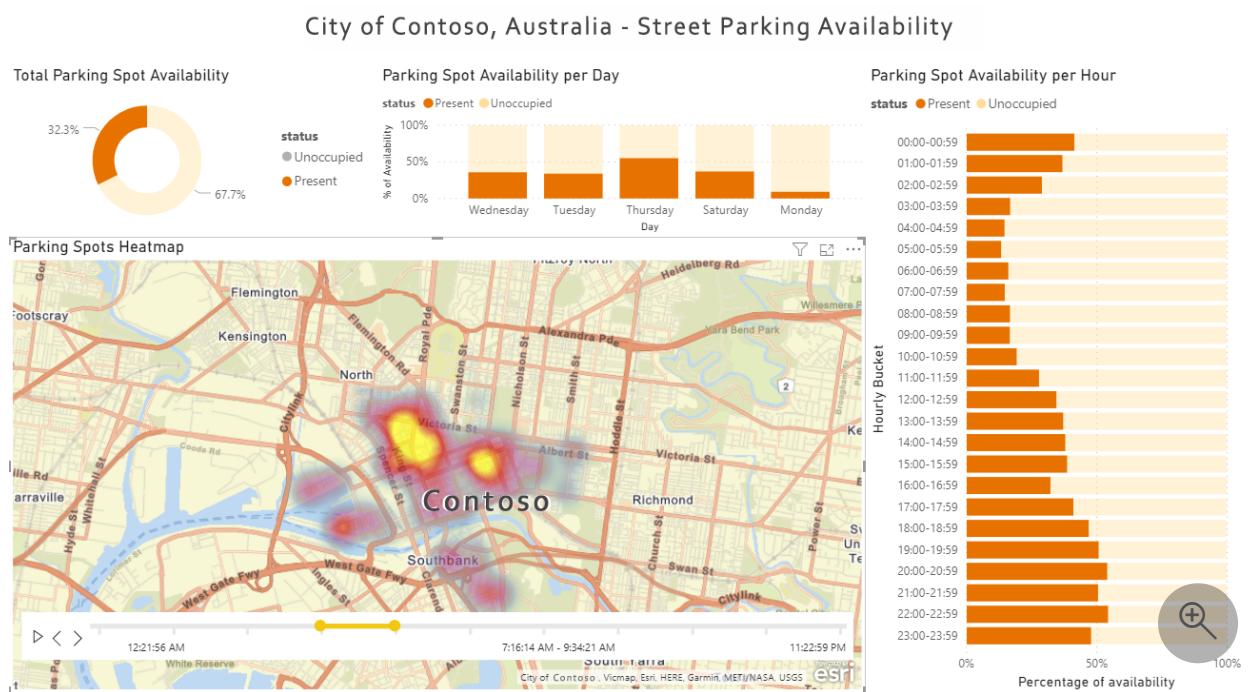
## Solution requirements

- Ability to collect data from different sources or systems.
- Infrastructure as code: deploy new dev and staging (stg) environments in an automated manner.
- Deploy application changes across different environments in an automated manner:
  - Implement Continuous Integration/Continuous Delivery (CI/CD) pipelines.
  - Use deployment gates for manual approvals.
- Pipeline as Code: ensure the CI/CD pipeline definitions are in source control.
- Carry out integration tests on changes using a sample data set.
- Run pipelines on a scheduled basis.
- Support future agile development, including the addition of data science workloads.
- Support for both row-level and object-level security:
  - The security feature is available in SQL Database.
  - You can also find it in Azure Synapse Analytics, Azure Analysis Services (AAS) and Power BI.
- Support for 10 concurrent dashboard users and 20 concurrent power users.

- The data pipeline should carry out data validation and filter out malformed records to a specified store.
- Support monitoring.
- Centralized configuration in a secure storage like Azure Key Vault.

## Potential use cases

This article uses the fictional city of Contoso to describe the use case scenario. In the narrative, Contoso owns and manages parking sensors for the city. It also owns the APIs that connect to and get data from the sensors. They need a platform that will collect data from many different sources. The data then must be validated, cleansed, and transformed to a known schema. Contoso city planners can then explore and assess report data on parking use with data visualization tools, like Power BI, to determine whether they need more parking or related resources.



## Considerations

The following list summarizes key learnings and best practices demonstrated by this solution:

### Note

Each item in the list below links out to the related **Key Learnings** section in the docs for the parking sensor solution example on GitHub.

- Use Data Tiering in your Data Lake [↗](#).
- Validate data early in your pipeline [↗](#).
- Make your data pipelines replayable and idempotent [↗](#).
- Ensure data transformation code is testable [↗](#).
- Have a CI/CD pipeline [↗](#).
- Secure and centralize configuration [↗](#).
- Monitor infrastructure, pipelines, and data [↗](#).

## Deploy this scenario

The following list contains the high-level steps required to set up the Parking Sensors solution with corresponding Build and Release Pipelines. You can find detailed setup steps and prerequisites in this [Azure Samples repository](#) [↗](#).

## Setup and deployment

- 1. Initial setup:** Install any prerequisites, import the Azure Samples GitHub repository into your own repository, and set required environment variables.
- 2. Deploy Azure resources:** The solution comes with an automated deployment script. It deploys all necessary Azure resources and Microsoft Entra service principals per environment. The script also deploys Azure Pipelines, variable groups, and service connections.
- 3. Set up git integration in dev Data Factory:** Configure git integration to work with the imported GitHub repository.
- 4. Carry out an initial build and release:** Create a sample change in Data Factory, like enabling a schedule trigger, then watch the change automatically deploy across environments.

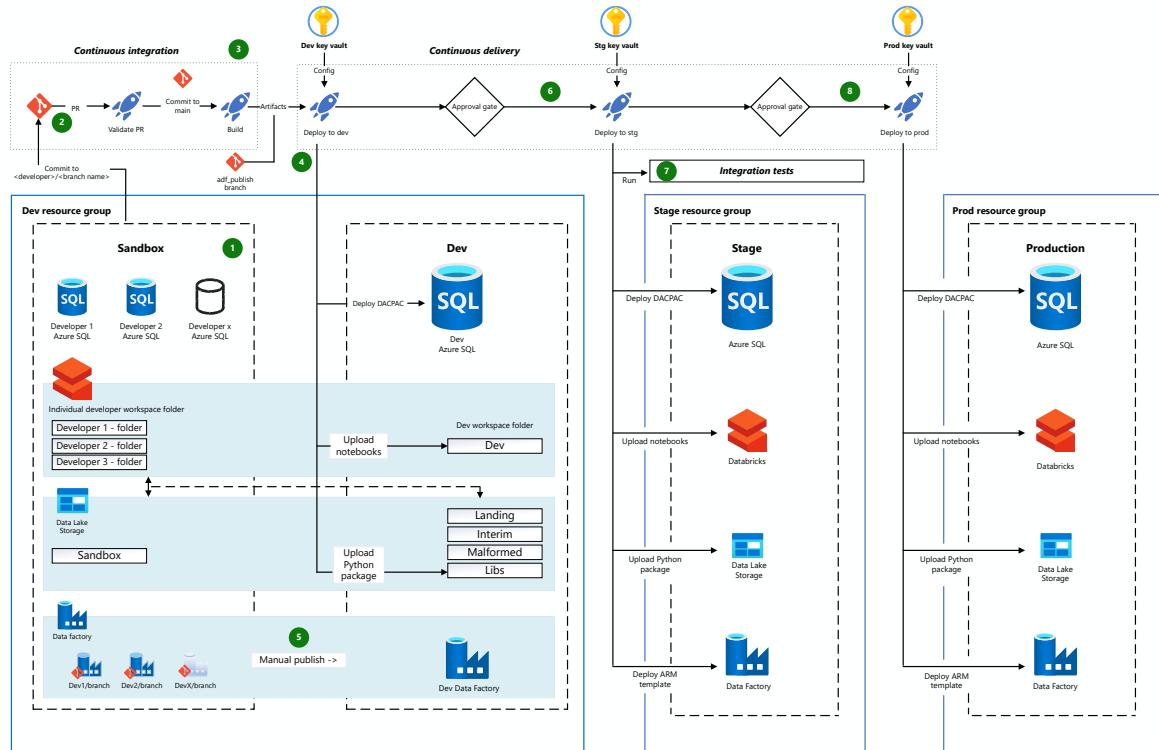
## Deployed resources

If deployment is successful, there should be three resources groups in Azure representing three environments: dev, stg, and prod. There should also be end-to-end build and release pipelines in Azure DevOps that can automatically deploy changes across these three environments.

For a detailed list of all resources, see the [Deployed Resources](#) section of the [DataOps - Parking Sensor Demo](#) README.

## Continuous integration and continuous delivery

The diagram below demonstrates the CI/CD process and sequence for the build and release pipelines.



Download a [Visio file](#) of this architecture.

1. Developers develop in their own sandbox environments within the dev resource group and commit changes into their own short-lived git branches. For example, `<developer_name>/<branch_name>`.
2. When changes are complete, developers raise a pull request (PR) to the main branch for review. Doing so automatically kicks-off the PR validation pipeline, which runs the unit tests, linting, and data-tier application package (DACPAC) builds.
3. On completion of the PR validation, the commit to main will trigger a build pipeline that publishes all necessary build artifacts.

4. The completion of a successful build pipeline will trigger the first stage of the release pipeline. Doing so deploys the publish build artifacts into the dev environment, except for ADF.

Developers manually publish to the dev ADF from the collaboration branch (main). The manual publishing updates the Azure Resource Manager (ARM) templates in the `adf_publish` branch.

5. The successful completion of the first stage triggers a manual approval gate.

On Approval, the release pipeline continues with the second stage, deploying changes to the `stg` environment.

6. Run integration tests to test changes in the `stg` environment.

7. Upon successful completion of the second stage, the pipeline triggers a second manual approval gate.

On Approval, the release pipeline continues with the third stage, deploying changes to the `prod` environment.

For more information, read the [Build and Release Pipeline](#) section of the README.

## Testing

The solution includes support for both unit testing and integration testing. It uses `pytest-adf` and the Nutter Testing Framework. For more information, see the [Testing](#) section of the README.

## Observability and monitoring

The solution supports observability and monitoring for Databricks and Data Factory. For more information, see the [Observability/Monitoring](#) section of the README.

## Next steps

If you'd like to deploy the solution, follow the steps in the [How to use the sample](#) section of the **DataOps - Parking Sensor Demo** README.

## Solution code samples on GitHub

- [Visit the project page on GitHub](#)

# Observability/monitoring

## Azure Databricks

- Monitoring Azure Databricks with Azure Monitor
- Monitoring Azure Databricks Jobs with Application Insights

## Data Factory

- Monitor Azure Data Factory with Azure Monitor
- Create alerts to proactively monitor your data factory pipelines ↗

## Synapse Analytics

- Monitoring resource utilization and query activity in Azure Synapse Analytics
- Monitor your Azure Synapse Analytics SQL pool workload using DMVs

## Azure Storage

- Monitor Azure Storage

# Resiliency and disaster recovery

## Azure Databricks

- Regional disaster recovery for Azure Databricks clusters

## Data Factory

- Create and configure a self-hosted integration runtime - High availability and scalability

## Synapse Analytics

- Geo-backups and Disaster Recovery
- Geo-restore for SQL Pool

## Azure Storage

- Disaster recovery and storage account failover
- Best practices for using Azure Data Lake Storage Gen2 – High availability and Disaster Recovery
- Azure Storage Redundancy

# Detailed walkthrough

For a detailed walk-through of the solution and key concepts, watch the following video recording: [DataDevOps for the Modern Data Warehouse on Microsoft Azure](#) ↗

## Related resources

- [Monitoring Azure Databricks with Azure Monitor](#)
- [Master data management with Profisee and Azure Data Factory](#)
- [DevTest and DevOps for PaaS solutions](#)

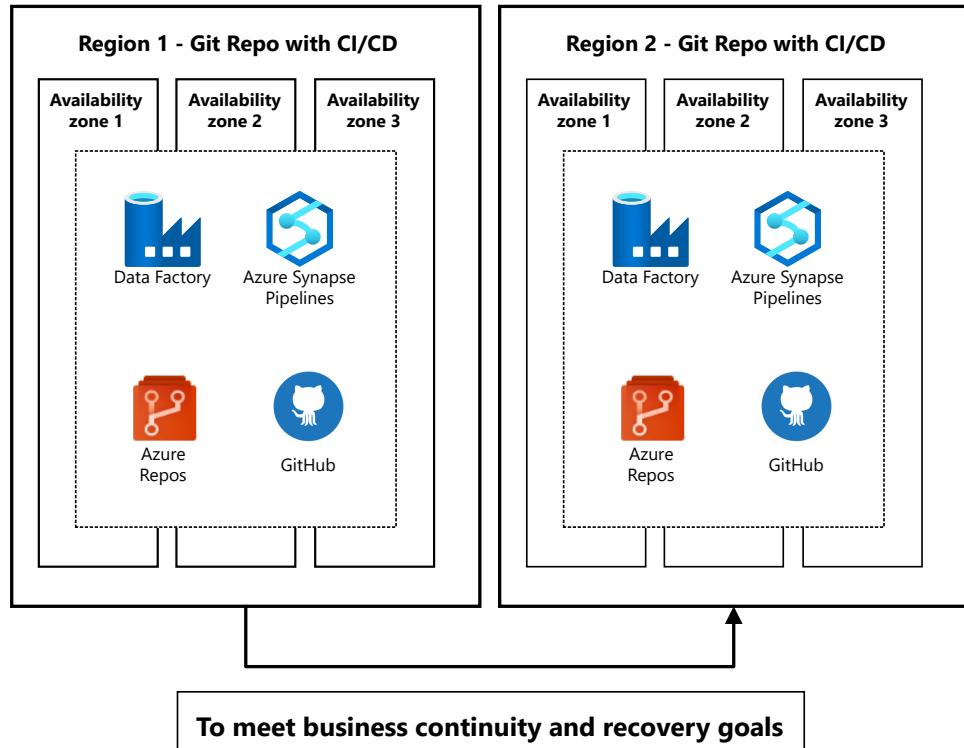
# BCDR for Azure Data Factory and Azure Synapse Analytics pipelines

Azure Data Factory   Azure Repos   Azure Synapse Analytics   GitHub

Disasters can be hardware failures, natural disasters, or software failures. The process of preparing for and recovering from a disaster is called disaster recovery (DR). This article discusses recommended practices to achieve business continuity and disaster recovery (BCDR) for Azure Data Factory and Azure Synapse Analytics pipelines.

BCDR strategies include availability zone redundancy, automated recovery provided by Azure disaster recovery, and user-managed recovery by using continuous integration/continuous delivery (CI/CD).

## Architecture



 Microsoft Azure



Download a [Visio file](#) of this architecture.

## Workflow

1. Data Factory and Azure Synapse pipelines achieve resiliency by using Azure regions and Azure availability zones.

- Each Azure region has a set of datacenters that are deployed within a latency-defined perimeter.
- Azure availability zones are physically separate locations within each Azure region that are tolerant to local failures.
- All Azure regions and availability zones are connected through a dedicated, regional low-latency network and by a high-performance network.
- All availability zone-enabled regions have at least three separate availability zones to ensure resiliency.

2. When a datacenter, part of a datacenter, or an availability zone in a region goes down, failover happens with zero downtime for zone-resilient Data Factory and Azure Synapse pipelines.

## Components

- [Azure Data Factory](#)
- [Azure Synapse Analytics](#) and [Azure Synapse pipelines](#)
- [GitHub](#)
- [Azure Repos](#)

## Scenario details

Data Factory and Azure Synapse pipelines store artifacts that include the following data:

### Metadata

- Pipeline
- Datasets
- Linked services
- Integration runtime
- Triggers

### Monitoring data

- Pipeline
- Triggers
- Activity runs

Disasters can strike in different ways, such as hardware failures, natural disasters, or software failures that result from human error or cyberattack. Depending on the types of

failures, their geographical impact can be regional or global. When planning a disaster recovery strategy, consider both the nature of the disaster and its geographic impact.

BCDR in Azure works on a shared responsibility model. Many Azure services require customers to explicitly set up their DR strategy, while Azure provides the baseline infrastructure and platform services as needed.

You can use the following recommended practices to achieve BCDR for Data Factory and Azure Synapse pipelines under various failure scenarios. For implementation, see [Deploy this scenario](#).

## Automated recovery with Azure disaster recovery

With automated recovery provided Azure backup and disaster recovery, when there is a complete regional outage for an Azure region that has a paired region, Data Factory or Azure Synapse pipelines automatically fail over to the paired region when you [Set up automated recovery](#). The exceptions are Southeast Asia and Brazil regions, where data residency requirements require data to stay in those regions.

In DR failover, Data Factory recovers the production pipelines. If you need to validate your recovered pipelines, you can back up the Azure Resource Manager (ARM) templates for your production pipelines in secret storage, and compare the recovered pipelines to the backups.

The Azure Global team conducts regular BCDR drills, and Azure Data Factory and Azure Synapse Analytics participate in these drills. The BCDR drill simulates a region failure and fails over Azure services to a paired region without any customer involvement. For more information about the BCDR drills, see [Testing of services](#).

## User-managed redundancy with CI/CD

To achieve BCDR in the event of an entire region failure, you need a data factory or an Azure Synapse workspace in the secondary region. In case of accidental Data Factory or Azure Synapse pipeline deletion, outages, or internal maintenance events, you can use Git and CI/CD to recover the pipelines manually.

Optionally, you can use an active/passive implementation. The primary region handles normal operations and remains active, while the secondary DR region requires pre-planned steps, depending on specific implementation, to be promoted to primary. In this case, all the necessary configurations for infrastructure are available in the secondary region, but they aren't provisioned.

## Potential use cases

User-managed redundancy is useful in scenarios like:

- Accidental deletion of pipeline artifacts through human error.
- Extended outages or maintenance events that don't trigger BCDR because there's no disaster reported.

You can quickly move your production workloads to other regions and not be affected.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

## Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

Data Factory and Azure Synapse pipelines are mainstream Azure services that support availability zones, and they're designed to provide the right level of resiliency and flexibility along with ultra-low latency.

The user-managed recovery approach allows you to continue operating if there are any maintenance events, outages, or human errors in the primary region. By using CI/CD, the Data Factory and Azure Synapse pipelines can integrate to a Git repository and deploy to a secondary region for immediate recovery.

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

User-managed recovery integrates Data Factory with Git by using CI/CD, and optionally uses a secondary DR region that has all the necessary infrastructure configurations as a backup. This scenario might incur added costs. To estimate costs, use the [Azure pricing calculator](#).

For examples of Data Factory and Azure Synapse Analytics pricing, see:

- Understanding Azure Data Factory pricing through examples
- Azure Synapse Analytics pricing ↗

## Operational excellence

Operational excellence covers the operations processes that deploy an application and keep it running in production. For more information, see [Overview of the operational excellence pillar](#).

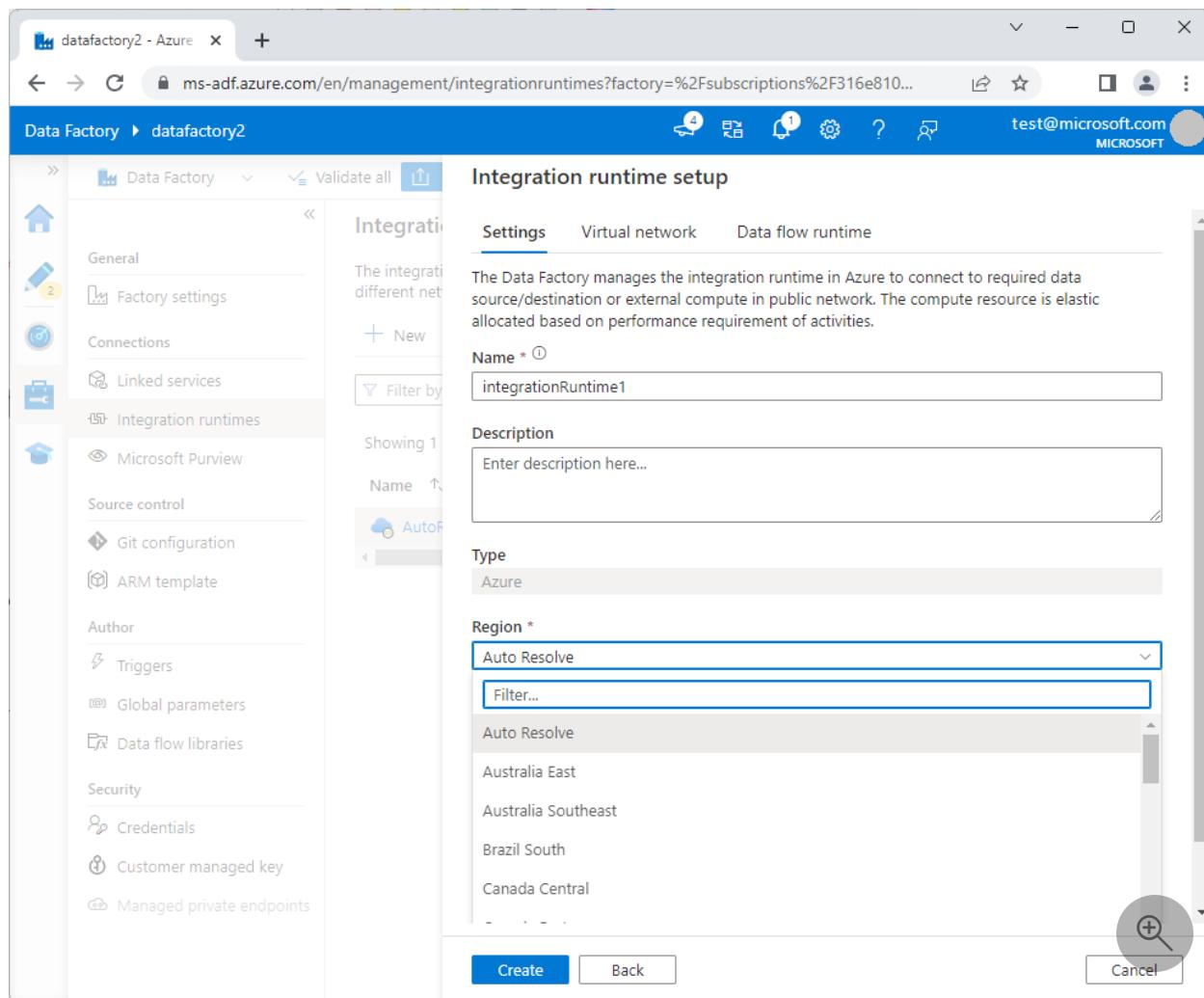
By using the user-managed CI/CD recovery approach, you can integrate to Azure Repos or GitHub. For more information about best CI/CD practices, see [Best practices for CI/CD](#).

## Deploy this scenario

Take the following actions to set up automated or user-managed DR for Data Factory and Azure Synapse pipelines.

### Set up automated recovery

In Data Factory, you can set the Azure integration runtime (IR) region for your activity execution or dispatch in the **Integration runtime setup**. To enable automatic failover in the event of a complete regional outage, set the **Region** to **Auto Resolve**.



The Data Factory manages the integration runtime in Azure to connect to required data source/destination or external compute in public network. The compute resource is elastic allocated based on performance requirement of activities.

**Name \*** integrationRuntime1

**Description** Enter description here...

**Type** Azure

**Region \*** Auto Resolve

Filter...

Auto Resolve

Australia East

Australia Southeast

Brazil South

Canada Central

**Create** **Back** **Cancel**

In the context of the integration runtimes, IR fails over automatically to the paired region when you select **Auto Resolve** as the IR region. For other specific location regions, you can create a secondary data factory in another region, and use CI/CD to provision your data factory from the Git repository.

- For managed virtual networks, Data Factory also automatically switches over to the managed IR.
- Azure managed automatic failover doesn't apply to self-hosted integration runtime (SHIR), because the infrastructure is customer-managed. For guidance on setting up multiple nodes for higher availability with SHIR, see [Create and configure a self-hosted integration runtime](#).
- To configure BCDR for Azure-SSIS IR, see [Configure Azure-SSIS integration runtime for business continuity and disaster recovery \(BCDR\)](#).

Linked services aren't fully enabled after failover, because of pending private endpoints in the newer network of the region. You need to configure private endpoints in the recovered region. You can automate private endpoint creation by using the [approval API](#).

# Set up user-managed recovery through CI/CD

You can use Git and CI/CD to recover pipelines manually in case of Data Factory or Azure Synapse pipeline deletion or outage.

- To use Data Factory pipeline CI/CD, see [Continuous integration and delivery in Azure Data Factory](#) and [Source control in Azure Data Factory](#).
- To use Azure Synapse pipeline CI/CD, see [Continuous integration and delivery for an Azure Synapse Analytics workspace](#). Make sure to initialize the Azure Synapse workspace first. For more information, see [Source control in Synapse Studio](#).

When you deploy user-managed redundancy by using CI/CD, take the following actions:

## Disable triggers

Disable triggers in the original primary data factory once it comes back online. You can disable the triggers manually, or implement automation to periodically check the availability of the original primary. Disable all triggers on the original primary data factory immediately after the factory recovers.

To use Azure PowerShell to turn Data Factory triggers off or on, see [Sample pre- and post-deployment script](#) and [CI/CD improvements related to pipeline triggers deployment](#).

## Handle duplicate writes

Most extract, transform, load (ETL) pipelines are designed to handle duplicate writes, because backfill and restatement require them. Data sinks that support transparent failover can handle duplicate writes with records merge or by deleting and inserting all records in the specific time range.

For data sinks that change endpoints after failover, primary and secondary storage might have duplicate or partial data. You need to merge the data manually.

## Check the witness and control the pipeline flow (optional)

In general, you need to design your pipelines to include activities, like fail and lookup activities, for restarting failed pipelines from the point of interest.

1. Add a global parameter in your data factory to indicate the region, for example `region='EastUS'` in the primary and `region='CentralUS'` in the secondary data factory.

2. Create a witness in a third region. The witness can be a REST call or any type of storage. The witness returns the current primary region, for example '`EastUS`', by default.
3. When a disaster happens, manually update the witness to return the new primary region, for example '`CentralUS`'.
4. Add an activity in your pipeline to look up the witness and compare the current primary value to the global parameter.
  - If the parameters match, this pipeline is running on the primary region. Proceed with the real work.
  - If the parameters don't match, this pipeline is running on the secondary region. Just return the result.

 **Note**

This approach introduces a dependency on the witness lookup into your pipeline. Failure to read the witness halts all pipeline runs.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal authors:

- [Krishnakumar Rukmangathan](#) | Senior Program Manager - Azure Data Factory team
- [Sunil Sabat](#) | Principal Program Manager - Azure Data Factory team

Other contributors:

- [Mario Zimmermann](#) | Principal Software Engineering Manager - Azure Data Factory team
- [Wee Hyong Tok](#) | Principal Director of PM - Azure Data Factory team

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- Business continuity management in Azure
- Resiliency in Azure
- Azure resiliency terminology
- Regions and availability zones
- Cross-region replication in Azure
- Azure regions decision guide
- Azure services that support availability zones
- Shared responsibility in the cloud
- Azure Data Factory data redundancy
- Integration runtime in Azure Data Factory
- Pipelines and activities in Azure Data Factory and Azure Synapse Analytics
- Data integration in Azure Synapse Analytics versus Azure Data Factory

## Related resources

- Enterprise-scale disaster recovery
- SMB disaster recovery with Azure Site Recovery
- Build high availability into your BCDR strategy
- High availability and disaster recovery scenarios for IaaS apps
- Choose a data pipeline orchestration technology in Azure
- Business continuity and disaster recovery for Azure Logic Apps

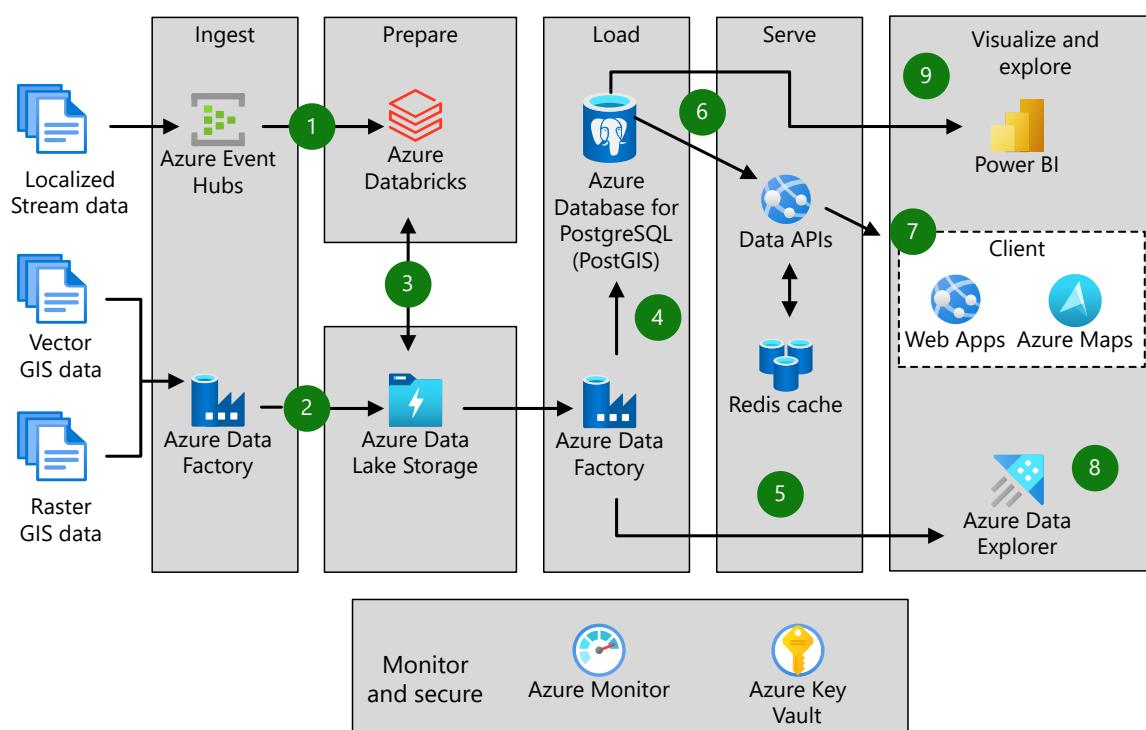
# Geospatial data processing and analytics

Azure Data Factory   Azure Data Lake Storage   Azure Database for PostgreSQL   Azure Databricks

Azure Event Hubs

This article outlines a manageable solution for making large volumes of geospatial data available for analytics.

## Architecture



Download a [Visio file](#) of this architecture.

The diagram contains several gray boxes, each with a different label. From left to right, the labels are Ingest, Prepare, Load, Serve, and Visualize and explore. A final box underneath the others has the label Monitor and secure. Each box contains icons that represent various Azure services. Numbered arrows connect the boxes in the way that the steps describe in the diagram explanation.

# Workflow

1. IoT data enters the system:
  - [Azure Event Hubs](#) ingests streams of IoT data. The data contains coordinates or other information that identifies locations of devices.
  - Event Hubs uses [Azure Databricks](#) for initial stream processing.
  - Event Hubs stores the data in [Azure Data Lake Storage](#).
2. GIS data enters the system:
  - [Azure Data Factory](#) ingests raster GIS data and vector GIS data of any format.
    - Raster data consists of grids of values. Each pixel value represents a characteristic like the temperature or elevation of a geographic area.
    - Vector data represents specific geographic features. Vertices, or discrete geometric locations, make up the vectors and define the shape of each spatial object.
  - Data Factory stores the data in Data Lake Storage.
3. Spark clusters in Azure Databricks use geospatial code libraries to transform and normalize the data.
4. Data Factory loads the prepared vector and raster data into [Azure Database for PostgreSQL](#). The solution uses the PostGIS extension with this database.
5. Data Factory loads the prepared vector and raster data into [Azure Data Explorer](#).
6. Azure Database for PostgreSQL stores the GIS data. APIs make this data available in standardized formats:
  - [GeoJSON](#) is based on JavaScript Object Notation (JSON). GeoJSON represents simple geographical features and their non-spatial properties.
  - [Well-known text \(WKT\)](#) is a text markup language that represents vector geometry objects.
  - [Vector tiles](#) are packets of geographic data. Their lightweight format improves mapping performance.
- A Redis cache improves performance by providing quick access to the data.
7. The Web Apps feature of [Azure App Service](#) works with Azure Maps to create visuals of the data.
8. Users analyze the data with Azure Data Explorer. GIS features of this tool create insightful visualizations. Examples include creating scatterplots from geospatial

data.

9. [Power BI](#) provides customized reports and business intelligence (BI). The Azure Maps visual for Power BI highlights the role of location data in business results.

Throughout the process:

- [Azure Monitor](#) collects information on events and performance.
- Log Analytics runs queries on Monitor logs and analyzes the results.
- [Azure Key Vault](#) secures passwords, connection strings, and secrets.

## Components

- [Azure Event Hubs](#) is a fully managed streaming platform for big data. This platform as a service (PaaS) offers a partitioned consumer model. Multiple applications can use this model to process the data stream at the same time.
- [Azure Data Factory](#) is an integration service that works with data from disparate data stores. You can use this fully managed, serverless platform to create, schedule, and orchestrate data transformation workflows.
- [Azure Databricks](#) is a data analytics platform. Its fully managed Spark clusters process large streams of data from multiple sources. Azure Databricks can transform geospatial data at large scale for use in analytics and data visualization.
- [Data Lake Storage](#) is a scalable and secure data lake for high-performance analytics workloads. This service can manage multiple petabytes of information while sustaining hundreds of gigabits of throughput. The data typically comes from multiple, heterogeneous sources and can be structured, semi-structured, or unstructured.
- [Azure Database for PostgreSQL](#) is a fully managed relational database service that's based on the community edition of the open-source [PostgreSQL](#) database engine.
- [PostGIS](#) is an extension for the PostgreSQL database that integrates with GIS servers. PostGIS can run SQL location queries that involve geographic objects.
- [Redis](#) is an open-source, in-memory data store. Redis caches keep frequently accessed data in server memory. The caches can then quickly process large volumes of application requests that use the data.
- [Power BI](#) is a collection of software services and apps. You can use Power BI to connect unrelated sources of data and create visuals of them.

- The [Azure Maps visual for Power BI](#) provides a way to enhance maps with spatial data. You can use this visual to show how location data affects business metrics.
- [Azure App Service](#) and its [Web Apps](#) feature provide a framework for building, deploying, and scaling web apps. The App Service platform offers built-in infrastructure maintenance, security patching, and scaling.
- [GIS data APIs in Azure Maps](#) store and retrieve map data in formats like GeoJSON and vector tiles.
- [Azure Data Explorer](#) is a fast, fully managed data analytics service that can work with [large volumes of data](#). This service originally focused on time series and log analytics. It now also handles diverse data streams from applications, websites, IoT devices, and other sources. [Geospatial functionality](#) in Azure Data Explorer provides options for rendering map data.
- [Azure Monitor](#) collects data on environments and Azure resources. This diagnostic information is helpful for maintaining availability and performance. Two data platforms make up Monitor:
  - [Azure Monitor Logs](#) records and stores log and performance data.
  - [Azure Monitor Metrics](#) collects numerical values at regular intervals.
- [Log Analytics](#) is an Azure portal tool that runs queries on Monitor log data. Log Analytics also provides features for charting and statistically analyzing query results.
- [Key Vault](#) stores and controls access to secrets such as tokens, passwords, and API keys. Key Vault also creates and controls encryption keys and manages security certificates.

## Alternatives

- Instead of developing your own APIs, consider using [Martin](#). This open-source tile server makes vector tiles available to web apps. Written in [Rust](#), Martin connects to PostgreSQL tables. You can deploy it as a container.
- If your goal is to provide a standardized interface for GIS data, consider using [GeoServer](#). This open framework implements industry-standard [Open Geospatial Consortium \(OGC\)](#) protocols such as [Web Feature Service \(WFS\)](#). It also integrates with common spatial data sources. You can deploy GeoServer as a container on a virtual machine. When customized web apps and exploratory queries are secondary, GeoServer provides a straightforward way to publish geospatial data.

- Various Spark libraries are available for working with geospatial data on Azure Databricks. This solution uses these libraries:
    - [Apache Sedona \(GeoSpark\)](#)
    - [GeoPandas](#)
- But [other solutions also exist for processing and scaling geospatial workloads with Azure Databricks](#).
- [Vector tiles](#) provide an efficient way to display GIS data on maps. This solution uses PostGIS to dynamically query vector tiles. This approach works well for simple queries and result sets that contain well under 1 million records. But in the following cases, a different approach may be better:
    - Your queries are computationally expensive.
    - Your data doesn't change frequently.
    - You're displaying large data sets.

In these situations, consider using [Tippecanoe](#) to generate vector tiles. You can run Tippecanoe as part of your data processing flow, either as a container or with [Azure Functions](#). You can make the resulting tiles available through APIs.

- Like Event Hubs, [Azure IoT Hub](#) can ingest large amounts of data. But IoT Hub also offers bi-directional communication capabilities with devices. If you receive data directly from devices but also send commands and policies back to devices, consider IoT Hub instead of Event Hubs.
- To streamline the solution, omit these components:
  - Azure Data Explorer
  - Power BI

## Scenario details

Many possibilities exist for working with *geospatial data*, or information that includes a geographic component. For instance, geographic information system (GIS) software and standards are widely available. These technologies can store, process, and provide access to geospatial data. But it's often hard to configure and maintain systems that work with geospatial data. You also need expert knowledge to integrate those systems with other systems.

This article outlines a manageable solution for making large volumes of geospatial data available for analytics. The approach is based on [Advanced Analytics Reference Architecture](#) and uses these Azure services:

- Azure Databricks with GIS Spark libraries processes data.

- Azure Database for PostgreSQL queries data that users request through APIs.
- Azure Data Explorer runs fast exploratory queries.
- Azure Maps creates visuals of geospatial data in web applications.
- The Azure Maps Power BI visual feature of Power BI provides customized reports

## Potential use cases

This solution applies to many areas:

- Processing, storing, and providing access to large amounts of raster data, such as maps or climate data.
- Identifying the geographic position of enterprise resource planning (ERP) system entities.
- Combining entity location data with GIS reference data.
- Storing Internet of Things (IoT) telemetry from moving devices.
- Running analytical geospatial queries.
- Embedding curated and contextualized geospatial data in web apps.

## Considerations

The following considerations, based on the [Microsoft Azure Well-Architected Framework](#), apply to this solution.

## Availability

- [Event Hubs spreads failure risk across clusters.](#)
  - Use a namespace with availability zones turned on to spread risk across three physically separated facilities.
  - Consider using the geo-disaster recovery feature of Event Hubs. This feature replicates the entire configuration of a namespace from a primary to a secondary namespace.
- See [business continuity features that Azure Database for PostgreSQL offers](#). These features cover a range of recovery objectives.
- [App Service diagnostics](#) alerts you to problems in apps, such as downtime. Use this service to identify, troubleshoot, and resolve issues like outages.
- Consider using [App Service to back up application files](#). But be careful with backed-up files, which include app settings in plain text. Those settings can contain secrets like connection strings.

# Scalability

This solution's implementation meets these conditions:

- Processes up to 10 million data sets per day. The data sets include batch or streaming events.
- Stores 100 million data sets in an Azure Database for PostgreSQL database.
- Queries 1 million or fewer data sets at the same time. A maximum of 30 users run the queries.

The environment uses this configuration:

- An Azure Databricks cluster with four F8s\_V2 worker nodes.
- A memory-optimized instance of Azure Database for PostgreSQL.
- An App Service plan with two Standard S2 instances.

Consider these factors to determine which adjustments to make for your implementation:

- Your data ingestion rate.
- Your volume of data.
- Your query volume.
- The number of parallel queries you need to support.

You can scale Azure components independently:

- Event Hubs automatically scales up to meet usage needs. But take steps to [manage throughput units](#) and [optimize partitions](#).
- Data Factory handles large amounts of data. Its [serverless architecture supports parallelism at different levels](#).
- [Data Lake Storage is scalable by design](#).
- Azure Database for PostgreSQL offers [high-performance horizontal scaling](#).
- [Azure Databricks clusters resize as needed](#).
- [Azure Data Explorer can elastically scale to terabytes of data in minutes](#).
- [App Service web apps scale up and out](#).

The [autoscale feature of Monitor](#) also provides scaling functionality. You can configure this feature to add resources to handle increases in load. It can also remove resources to save money.

# Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

- Protect vector tile data. Vector tiles embed coordinates and attributes for multiple entities in one file. If you generate vector tiles, use a dedicated set of tiles for each permission level in your access control system. With this approach, only users within each permission level have access to that level's data file.
- To improve security, use Key Vault in these situations:
  - [To manage keys that Event Hubs uses to encrypt data](#).
  - [To store credentials that Data Factory uses in pipelines](#).
  - [To secure application settings and secrets that your App Service web app uses](#).
- See [Security in Azure App Service](#) for information on how App Service helps secure web apps. Also consider these points:
  - See how to [get the certificate that your app needs if it uses a custom domain name](#).
  - See how to [redirect HTTP requests for your app to the HTTPS port](#).
  - Learn about [best practices for authentication in web apps](#).

# Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

- To estimate the cost of implementing this solution, see a sample [cost profile](#). This profile is for a single implementation of the environment described in [Scalability considerations](#). It doesn't include the cost of Azure Data Explorer.
- To adjust the parameters and explore the cost of running this solution in your environment, use the [Azure pricing calculator](#).

# Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Richard Bumann](#) | Solution Architect

# Next steps

Product documentation:

- [About Azure Event Hubs](#)
- [Azure Databricks concepts](#)
- [Introduction to Azure Data Lake Storage](#)
- [What is Azure Data Factory?](#)
- [Azure App Service overview](#)

To start implementing this solution, see this information:

- [Connect a WFS to Azure Maps](#)
- [Process OpenStreetMap data ↗ with Spark.](#)
- [Explore ways to display data with Azure Maps.](#)

## Information on processing geospatial data

- [Functions for querying PostGIS for vector tiles ↗](#)
- [Functions for loading PostGIS rasters ↗](#)
- [Azure Data Explorer geospatial functions](#)
- [Data sources for vector tiles in Azure Maps](#)
- [Approaches for processing geospatial data in Databricks ↗](#)

## Related resources

### Related architectures

- [Big data analytics with Azure Data Explorer](#)
- [Health data consortium on Azure](#)
- [\[DataOps for the modern data warehouse\]\[DataOps for the modern data warehouse\]](#)
- [Azure Data Explorer interactive analytics](#)
- [Geospatial reference architecture - Azure Orbital](#)
- [Geospatial analysis for telecom](#)
- [Spaceborne data analysis with Azure Synapse Analytics](#)

### Related guides

- [Compare the machine learning products and technologies from Microsoft - Azure Databricks](#)

- Machine learning operations (MLOps) framework to scale up machine learning lifecycle with Azure Machine Learning
- [Azure Machine Learning decision guide for optimal tool selection][Azure Machine Learning decision guide for optimal tool selection]
- Monitor Azure Databricks

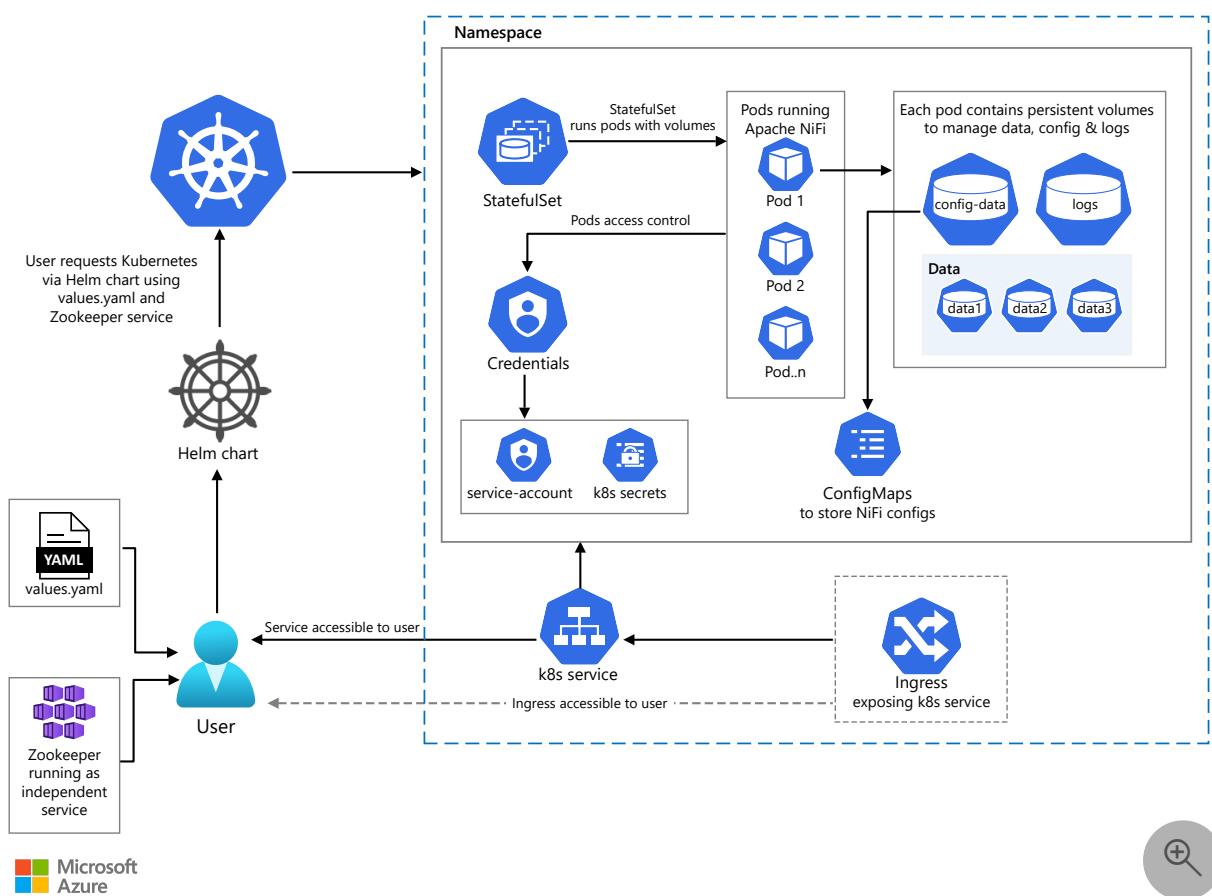
# Helm-based deployments for Apache NiFi

Azure Kubernetes Service (AKS)

This solution shows you how to use Helm charts when you deploy NiFi on Azure Kubernetes Service (AKS). Helm streamlines the process of installing and managing Kubernetes applications.

*Apache®, Apache NiFi®, and NiFi® are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.*

## Architecture



Download a [Visio file](#) of this architecture.

## Workflow

- A Helm chart contains a `values.yaml` file. That file lists input values that users can edit.
- A user adjusts settings in a chart, including values for:
  - Volume sizes.
  - The number of pods.
  - User authentication and authorization mechanisms.
- The user runs the Helm `install` command to deploy the chart.
- Helm checks whether the user input contains values for all required variables.
- Helm creates a manifest that describes the objects to deploy on Kubernetes.
- Helm sends the manifest to the Kubernetes cluster. Apache ZooKeeper provides cluster coordination.
- Kubernetes creates the specified objects. A NiFi deployment requires these objects:
  - Configuration objects.
  - Data volumes. Pod storage is temporary.
  - A log volume.
  - Pods that use an image to run NiFi in a container. Kubernetes uses a *StatefulSet* workload resource to manage the pods.
  - A Kubernetes service that makes the NiFi UI available to users.
  - Ingress routes if the cluster uses ingress to make the UI available externally.

## Components

A Helm chart is a collection of files in a folder with a tree structure. These files describe Kubernetes resources. You can configure the following components in a Helm chart:

### ZooKeeper

ZooKeeper uses a separate chart. You can use the standard ZooKeeper chart that Kubernetes supplies in its [incubator chart repository](#). But when your dependencies include public registry content, you introduce risk into your image development and deployment workflows. To mitigate this risk, keep local copies of public content when you can. For detailed information, see [Manage public content with Azure Container Registry](#).

As an alternative, you can deploy ZooKeeper on your own. If you choose this option, provide the ZooKeeper server and port number so that the pods that run NiFi can access the ZooKeeper service.

## Kubernetes StatefulSet

To run an application on Kubernetes, you run a pod. This basic unit runs different containers that implement the application's different activities.

Kubernetes offers two solutions for managing pods that run an application like NiFi:

- A *ReplicaSet*, which maintains a stable set of the replica pods that run at any given time. You often use a ReplicaSet to guarantee the availability of a specified number of identical pods.
- A *StatefulSet*, which is the workload API object that you use to manage stateful applications. A StatefulSet manages pods that are based on an identical container specification. Kubernetes creates these pods from the same specification. But these pods aren't interchangeable. Each pod has a persistent identifier that it maintains across rescheduling.

Since you use NiFi to manage data, a StatefulSet provides the best solution for NiFi deployments.

## ConfigMaps

Kubernetes offers *ConfigMaps* for storing non-confidential data. Kubernetes uses these objects to manage various configuration files like `nifi.properties`. The container that runs the application accesses the configuration information through mounted volumes and files. ConfigMaps make it easy to manage post-deployment configuration changes.

## ServiceAccount

In secured instances, NiFi uses authentication and authorization. NiFi manages this information in file system files. Specifically, each cluster node needs to maintain an `authorizations.xml` file and a `users.xml` file. All members need to be able to write to these files. And each node in the cluster needs to have an identical copy of this information. Otherwise, the cluster goes out of sync and breaks down.

To meet these conditions, you can copy these files from the first member of the cluster to every member that comes into existence. Each new member then maintains its own copies. Pods generally don't have authorization to copy content from another pod. But a Kubernetes *ServiceAccount* provides a way to get authorization.

## Services

Kubernetes services make the application service available to users of the Kubernetes cluster. Service objects also make it possible for member nodes of NiFi clusters to communicate with each other. For Helm chart deployments, use two service types: headless services and IP-based services.

## Ingress

An ingress manages external access to cluster services. Specifically, a pre-configured ingress controller exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. You can define ingress rules that determine how the controller routes the traffic. The Helm chart includes the ingress route in the configuration.

## Secrets

To configure secured NiFi clusters, you need to store credentials. Kubernetes secrets provide a secure way to store and retrieve these credentials.

## Scenario details

[Apache NiFi](#) users often need to deploy NiFi on Kubernetes. A Kubernetes deployment involves many objects, such as pods, volumes, and services. It's difficult to manage the *manifests*, or specification files, that Kubernetes uses for this number of objects. The difficulty increases when you deploy several NiFi clusters that use different configurations.

Helm *charts* provide a solution for managing the manifests. Helm is the package manager for Kubernetes. By using the Helm tool, you can streamline the process of installing and managing Kubernetes applications.

A chart is the packaging format that Helm uses. You enter configuration requirements into chart files. Helm keeps track of each chart's history and versions. Helm then uses charts to generate Kubernetes manifest files.

From a single chart, you can deploy applications that use different configurations. When you run [NiFi on Azure](#), you can use Helm charts to deploy different NiFi configurations on Kubernetes.

Apache®, Apache NiFi®, and NiFi® are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

# Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

## Data disks

For disk usage, consider using a striped set of disks for repositories. In test deployments that used Virtual Machine Scale Sets, this approach worked best. The following excerpt from `nifi.properties` shows a disk usage configuration:

```
config

nifi.flowfile.repository.directory=/data/partition1/flowfiles
nifi.provenance.repository.directory.stripe1=/data/partition1/provenancenifi
.niifi.provenance.repository.directory.stripe2=/data/partition2/provenancenifi.pro
venance.repository.directory.stripe3=/data/partition3/provenancenifi.content
.repository.directory.stripe2=/data/partition2/content
nifi.content.repository.directory.stripe3=/data/partition3/content
```

This configuration uses three volumes of equal size. You can adjust the values and the striping to meet your system requirements.

## Deployment scenarios

You can use a public or private load balancer or an ingress controller to expose a NiFi cluster. When you use Helm charts for this implementation, two configurations are available:

- An unsecured NiFi cluster that's accessible through an HTTP URL without user authentication or authorization.
- A secured NiFi cluster that's accessible through an HTTPS URL. This kind of cluster is secured with TLS. When you configure secured clusters, you can provide your own certificates. Alternatively, the charts can generate the certificates. For this purpose, the charts use a NiFi toolkit that provides a self-signed Certificate Authority (CA).

If you configure a NiFi cluster to run as a secured cluster with TLS communication, you need to turn on user authentication. Use one of the following supported user authentication methods:

- Certificate-based user authentication. Users are authenticated by the certificate that they present to the NiFi UI. To use this kind of user authentication system, add the CA's public certificate to the NiFi deployment.
- LDAP-based user authentication. An LDAP server authenticates user credentials. When you deploy the chart, provide information about the LDAP server and the information tree.
- OpenID-based user authentication. Users provide information to the OpenID server to configure the deployment.

## Resource configuration and usage

To optimize resource usage, use these Helm options to configure CPU and memory values:

- The `request` option, which specifies the initial amount of the resource that the container requests
- The `limit` option, which specifies the maximum amount of the resource that the container can use

When you configure NiFi, consider your system's memory configuration. Because NiFi is a Java application, you should adjust settings like the minimum and maximum java virtual machine (JVM) memory values. Use the following settings:

- `jvmMinMemory`
- `jvmMaxMemory`
- `g1ReservePercent`
- `conGcThreads`
- `parallelGcThreads`
- `initiatingHeapOccupancyPercent`

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Use a Kubernetes security context to improve the security of the underlying containers that run the NiFi binary. A security context manages access to those containers and their pods. Through a security context, you can grant non-root users permissions to run the containers.

Other uses of security contexts include:

- Restricting the access of OS-based users that run the containers.
- Specifying which groups can access the containers.
- Limiting access to the file system.

## Container images

Kubernetes containers are the basic units that run NiFi binaries. To configure a NiFi cluster, focus on the image that you use to run these containers. You have two options for this image:

- Use the standard NiFi image to run the NiFi chart. The Apache NiFi community supplies that image. But you need to add a `kubectl` binary to the containers to configure secured clusters.
- Use a custom image. If you take this approach, consider your file system requirements. Ensure that the location of your NiFi binaries is correct. For more information on the configured file system, see [Dockerfile in the Apache NiFi source code](#).

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Muazma Zahid](#) | Principal PM Manager

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [Helm](#)
- [Helm charts](#)
- [Kubernetes](#)
- [Kubernetes StatefulSets](#)
- [Kubernetes Volumes](#)
- [Kubernetes ConfigMaps](#)
- [Kubernetes Secrets](#)
- [Kubernetes Service](#)
- [Kubernetes Ingress](#)
- [Azure Kubernetes Service](#)

- [Apache NiFi Docker Image ↗](#)

## Related resources

- [Apache NiFi on Azure](#)
- [Apache NiFi monitoring with MonitoFi](#)
- [Microservices architecture on Azure Kubernetes Service \(AKS\)](#)
- [Advanced Azure Kubernetes Service \(AKS\) microservices architecture](#)

# Master data management with Profisee and Azure Data Factory

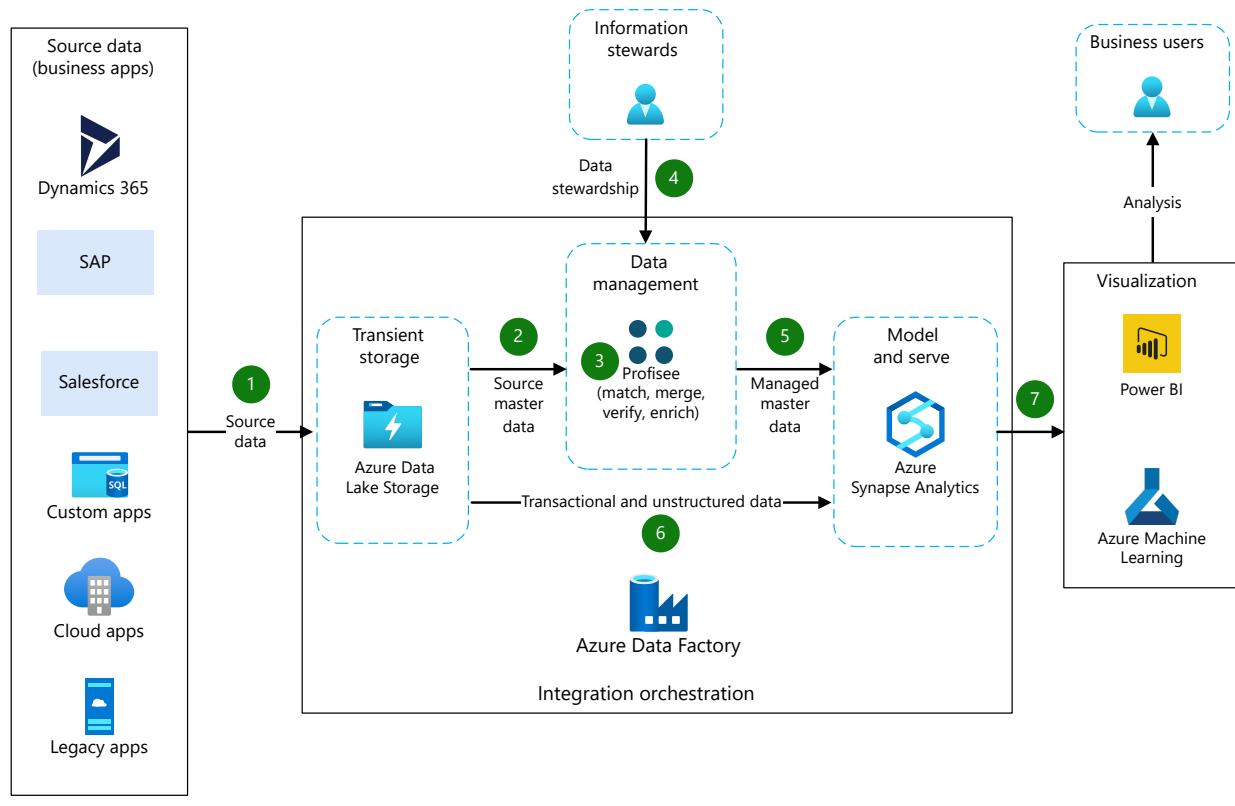
Azure Data Factory   Azure Databricks   Azure Data Lake

This architectural pattern demonstrates how you can incorporate MDM into the Azure data services ecosystem to improve the quality of data used for analytics and operational decision making. MDM solves several common challenges, including:

- Identifying and managing duplicate data (match and merge).
- Flagging and resolving data quality issues.
- Standardizing and enriching data.
- Allowing data stewards to proactively manage and improve the data.

This pattern presents a modern approach to MDM. All technologies are deployable natively in Azure, including Profisee, which you can deploy via containers and manage with Azure Kubernetes Service.

## Architecture



Download a [Visio file](#) of the diagrams used in this architecture.

## Dataflow

The following dataflow corresponds to the preceding diagram:

1. **Source data load:** Source data from business applications copies to Azure Data Lake and stores it for further transformation and use in downstream analytics. Source data typically falls into one of three categories:

- Structured master data – The information that describes customers, products, locations, and so on. Master data is low-volume, high-complexity, and changes slowly over time. It's often the data that organizations struggle the most with in terms of data quality.
- Structured transactional data – Business events that occur at a specific point in time, such as an order, invoice, or interaction. Transactions include the metrics for that transaction (like sales price) and references to master data (like the product and customer involved in a purchase). Transactional data is typically high-volume, low-complexity, and doesn't change over time.
- Unstructured data – Data that can include documents, images, videos, social media content, and audio. Modern analytics platforms can increasingly use unstructured data to learn new insights. Unstructured data is often associated with master data, such as a customer associated with a social media account, or a product associated with an image.

2. **Source master data load:** Master data from source business applications loads into the MDM application "as is", with complete lineage information and minimal transformations.

3. **Automated MDM processing:** The MDM solution uses automated processes to standardize, verify, and enrich data, such as address data. The solution also identifies data quality issues, groups duplicate records (like duplicate customers), and generates master records, also called "golden records".

4. **Data stewardship:** As necessary, data stewards can:

- Review and manage groups of matched records
- Create and manage data relationships
- Fill in missing information
- Resolve data quality issues.

Data stewards can manage multiple alternate hierarchical roll-ups as required, such as product hierarchies.

5. **Managed master data load:** High-quality master data flows into downstream analytics solutions. This action simplifies the process since data integrations no longer require any data quality transformations.

6. **Transactional and unstructured data load:** Transactional and unstructured data loads into the downstream analytics solution where it combines with high-quality master data.

7. **Visualization and analysis:** Data is modeled and made available to business users for analysis. High-quality master data eliminates common data-quality issues, which result in improved insights.

## Components

- [Azure Data Factory](#) is a hybrid data integration service that lets you create, schedule, and orchestrate your ETL and ELT workflows.
- [Azure Data Lake](#) provides limitless storage for analytics data.
- [Profisee](#) is a scalable MDM platform that's designed to easily integrate with the Microsoft ecosystem.
- [Azure Synapse Analytics](#) is the fast, flexible, and trusted cloud data warehouse that lets you scale, compute, and store data elastically and independently, with a massively parallel processing architecture.
- [Power BI](#) is a suite of business analytics tools that delivers insights throughout your organization. Connect to hundreds of data sources, simplify data prep, and drive improvised analysis. Produce beautiful reports, then publish them for your organization to consume on the web and across mobile devices.

## Alternatives

Absent a purpose-built MDM application, you can find some of the technical capabilities needed to build an MDM solution within the Azure ecosystem.

- Data quality - When loading to an analytics platform, you can build data quality into the integration processes. For example, apply data quality transformations in an [Azure Data Factory](#) pipeline with hardcoded scripts.
- Data standardization and enrichment - [Azure Maps](#) helps provide data verification and standardization for address data, which you can use in Azure Functions and Azure Data Factory. Standardization of other data might require development of hardcoded scripts.

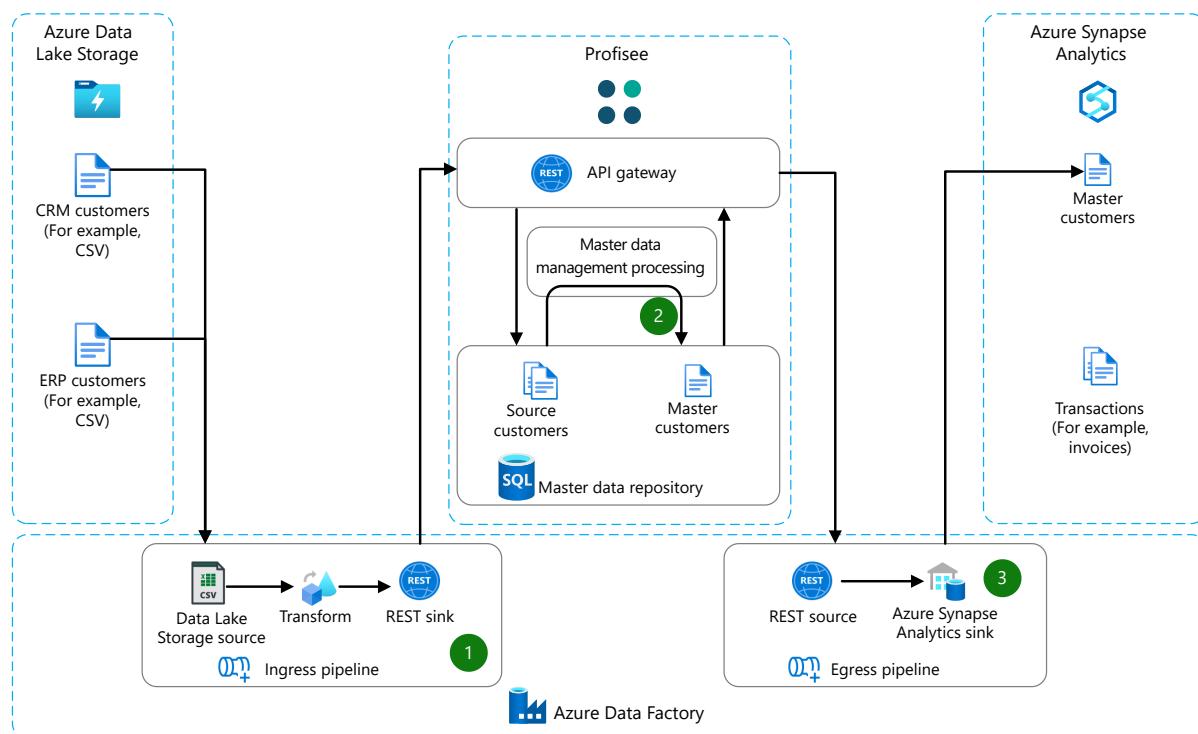
- Duplicate data management - You can use Azure Data Factory to [deduplicate rows](#) where sufficient identifiers are available for an exact match. In this case, the logic to merge matched with appropriate survivorship would likely require custom hardcoded scripts.
- Data stewardship - Use [Power Apps](#) to quickly develop simple data stewardship solutions to manage data in Azure, along with appropriate user interfaces for review, workflow, alerts, and validations.

## Scenario details

Many digital transformation programs use Azure as the core. But it depends on the quality and consistency of data from multiple sources, like business applications, databases, data feeds, and so on. It also delivers value through business intelligence, analytics, machine learning, and more. Profisee's Master Data Management (MDM) solution completes the Azure data estate with a practical method to "align and combine" data from multiple sources. It does so by enforcing consistent data standards on source data, like match, merge, standardize, verify, and correct. Native integration with Azure Data Factory and other Azure Data Services further streamlines this process to accelerate the delivery of Azure business benefits.

A core aspect of how MDM solutions function is that they combine data from multiple sources to create a "golden record master" that contains the best-known and trusted data for each record. This structure builds out domain-by-domain according to requirements, but it almost always requires multiple domains. Common domains are customer, product, and location. But domains can represent anything from reference data to contracts and drug names. In general, the better domain coverage that you can build out relative to the broad Azure data requirements the better.

## MDM integration pipeline



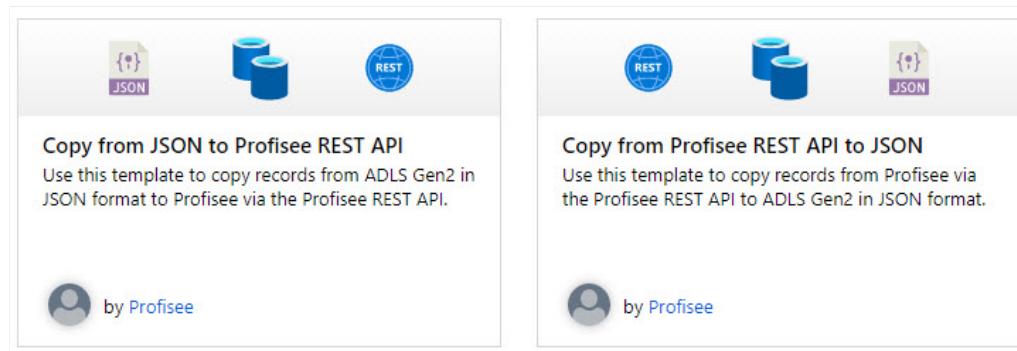
Download a [Visio file](#) of this architecture.

The preceding image shows the details for integrating with the Profisee MDM solution. Notice that Azure Data Factory and Profisee include native REST integration support, providing a lightweight and modern integration.

1. **Load source data to MDM:** Azure Data Factory extracts data from the data lake, transforms it to match the master data model, and streams it into the MDM repository via a REST sink.
2. **MDM processing:** The MDM platform processes source master data through a sequence of activities to verify, standardize, and enrich the data, and to execute data-quality processes. Finally, MDM performs matching and survivorship to identify and group duplicate records and create master records. Optionally, data stewards can perform tasks that result in a set of master data for use in downstream analytics.
3. **Load master data for analytics:** Azure Data Factory uses its REST source to stream master data from Profisee to Azure Synapse Analytics.

## Azure Data Factory templates for Profisee

In collaboration with Microsoft, Profisee has developed a set of Azure Data Factory templates that make it faster and easier to integrate Profisee into the Azure Data Services ecosystem. These templates use Azure Data Factories REST data source and data sink to read and write data from Profisee's REST Gateway API. They provide templates for both reading from and writing to Profisee.



## Example Data Factory template: JSON to Profisee over REST

The following screenshots show an Azure Data Factory template that copies data from a JSON file in an Azure Data Lake to Profisee via REST.

The template copies the source JSON data:

Save Save as template Validate Validate copy runtime Debug Trigger (1)

Copy data

Copy from JSON to REST API

General Source Sink Mapping Settings User properties

Source dataset \* InputBlobJson

▲ Dataset properties

NAME	VALUE	TYPE
FolderName	@pipeline().parameters.FileFolder	string
FileName	@pipeline().parameters.FileName	string

File path type  File path in dataset  Wildcard file path  List of files

Filter by last modified  Start time (UTC)  End time (UTC)

Recursively

Enable partition discovery

Max concurrent connections

Additional columns

Then, the data syncs to Profisee via REST:

Copy data

Copy from JSON to REST API

Sink dataset \* ProfiseeRESTAPI\_Members\_Update

Dataset properties

NAME	VALUE	TYPE
entityId	@replace(pipeline().parameters.FileFol...	string
isUpsert	true	bool

Request method \* PATCH

Request timeout 00:01:40

Request interval (ms) 10

Write batch size 10000

Http Compression type None

Additional headers

+ New | Delete

Name	Value
x-api-key	<example-key-value>

For more information, see [Azure Data Factory templates for Profisee](#).

## MDM processing

In an analytical MDM use case, data often processes through the MDM solution automatically to load data for analytics. The following sections show a typical process for customer data in this context.

### 1. Source data load

Source data loads into the MDM solution from source systems, including lineage information. In this case, we have two source records, one from CRM and one from the ERP application. Upon visual inspection, the two records appear to both represent the same person.

[ ] Expand table

Source Name	Source Address	Source State	Source Phone	Source ID	Standard Address	Standard State	Standard Name	Standard Phone	Similarity
Alana Bosh	123 Main Street	GA	7708434125	CRM-100					
Bosch, Alana	123 Main St.	Georgia	404-854-7736	CRM-121					

Source Name	Source Address	Source State	Source Phone	Source ID	Standard Address	Standard State	Standard Name	Standard Phone	Similarity
Alana Bosch		(404) 854-7736	ERP-988						

## 2. Data verification and standardization

Verification and standardization rules and services help standardize and verify address, name, and phone number information.

[Expand table](#)

Source Name	Source Address	Source State	Source Phone	Source ID	Standard Address	Standard State	Standard Name	Standard Phone	Similarity
Alana Bosh	123 Main Street	GA	7708434125	CRM-100	123 Main St.	GA	Alana Bosh	770 843 4125	
Bosch, Alana	123 Main St.	Georgia	404-854-7736	CRM-121	123 Main St.	GA	Alana Bosch	404 854 7736	
Alana Bosch		(404) 854-7736	ERP-988				Alana Bosch	404 854 7736	

## 3. Matching

With data standardized, matching occurs, identifying the similarity between records in the group. In this scenario, two records match each other exactly on Name and Phone, and the other fuzzy matches on Name and Address.

[Expand table](#)

Source Name	Source Address	Source State	Source Phone	Source ID	Standard Address	Standard State	Standard Name	Standard Phone	Similarity
Alana Bosh	123 Main Street	GA	7708434125	CRM-100	123 Main St.	GA	Alana Bosh	770 843 4125	0.9
Bosch, Alana	123 Main St.	Georgia	404-854-7736	CRM-121	123 Main St.	GA	Alana Bosch	404 854 7736	1.0
Alana Bosch		(404) 854-7736	ERP-988				Alana Bosch	404 854 7736	1.0

## 4. Survivorship

With a group formed, survivorship creates and populates a master record (also called a "golden record") to represent the group.

[Expand table](#)

Source Name	Source Address	Source State	Source Phone	Source ID	Standard Address	Standard State	Standard Name	Standard Phone	Similarity
Alana Bosh	123 Main Street	GA	7708434125	CRM-100	123 Main St.	GA	Alana Bosh	770 843 4125	0.9
Bosch, Alana	123 Main St.	Georgia	404-854-7736	CRM-121	123 Main St.	GA	Alana Bosch	404 854 7736	1.0
Alana Bosch	(404) 854-7736		ERP-988				Alana Bosch	404 854 7736	1.0
Master Record:	123 Main St.	GA	Alana Bosch	404 854 7736					

This master record, along with improved source data and lineage information, loads into the downstream analytics solution, where it links to transactional data.

This example shows basic, automated MDM processing. You can also use data quality rules to automatically calculate and update values, and flag missing or invalid values for data stewards to resolve. Data stewards help manage the data, including managing hierarchical rollups of data.

## The impact of MDM on integration complexity

As shown previously, MDM addresses several common challenges encountered when integrating data into an analytics solution. It includes correcting data quality issues, standardizing and enriching data, and rationalizing duplicate data. Incorporating MDM into your analytics architecture fundamentally changes the data flow by eliminating hardcoded logic in the integration process, and offloading it to the MDM solution, which significantly simplifies integrations. The following table outlines some common differences in the integration process with and without MDM.

[Expand table](#)

Capability	Without MDM	With MDM
Data quality	The integration processes include quality rules and transformations to help fix and correct data as it moves. It requires technical resources for both the initial implementation and ongoing maintenance of these rules, making data integration processes complicated and expensive to develop and maintain.	The MDM solution configures and enforces data quality logic and rules. Integration processes perform no data quality transformations, instead moving the data "as-is" into the MDM solution. Data integration processes are simple and affordable to develop and maintain.
Data standardization and enrichment	The integration processes include logic to standardize and align reference and master data. Develop integrations with third-party services to perform standardization of address, name, email, and phone data.	By using built-in rules and out-of-the-box integrations with third-party data services, you can standardize data within the MDM solution, which simplifies integration.
Duplicate data management	The integration process identifies and groups duplicate records that exist within and across applications based on existing unique identifiers. This process shares identifiers across systems (for example, SSN or email), and only matches and groups them when identical. More sophisticated approaches require significant investments in integration engineering.	Built-in machine learning matching capabilities identify duplicate records within and across systems, generating a golden record to represent the group. This process lets records be "fuzzy matched", grouping records that are similar, with explainable results. It manages groups in scenarios where the ML engine is unable to form a group with high confidence.
Data stewardship	Data stewardship activities only update data in the source applications, like ERP or CRM. Typically, they discover	MDM solutions have built-in data stewardship capabilities that let users access and manage data.

Capability	Without MDM	With MDM
	issues, like missing, incomplete, or incorrect data, when performing analytics. They correct the issues in the source application, and then update them in the analytics solution during the next update. Any new information to manage gets added to source applications, which takes time and is costly.	Ideally, the system flags issues and prompts data stewards to correct them. Quickly configure new information or hierarchies in the solution so that data stewards manage them.

## MDM use cases

While there are numerous use cases for MDM, a few use cases cover most real-world MDM implementations. Although these use cases focus on a single domain, they're unlikely built from only that domain. In other words, even these focused use cases most likely include multiple master data domains.

### Customer 360

Consolidating customer data for analytics is the most common MDM use case. Organizations capture customer data across an increasing number of applications, creating duplicate customer data within and across applications with inconsistencies and discrepancies. This poor-quality customer data makes it difficult to realize the value of modern analytics solutions. Symptoms include:

- Hard to answer basic business questions like "Who are our top customers?" and "How many new customers did we have?", requiring significant manual effort.
- Missing and inaccurate customer information, making it difficult to roll up or drill down into data.
- Inability to analyze customer data across systems or business units due to an inability to uniquely identify a customer across organizational and system boundaries.
- Poor-quality insights from AI and machine learning due to poor-quality input data.

### Product 360

Product data often spreads across multiple enterprise applications, such as ERP, PLM, or e-commerce. The result is a challenge understanding the total catalog of products that have inconsistent definitions for properties such as the product's name, description, and characteristics. And different definitions of reference data further complicate this situation. Symptoms include:

- Inability to support different alternative hierarchical rollup and drill-down paths for product analytics.
- Whether finished goods or material inventory, difficulty understanding exactly what products you have on hand, the vendors you purchase your products from, and duplicate products, leading to excess inventory.
- Difficulty rationalizing products due to conflicting definitions, which lead to missing or inaccurate information in analytics.

### Reference data 360

In the context of analytics, reference data exists as numerous lists of data that help further describe other sets of master data. Reference data can include lists of countries and regions, currencies, colors, sizes, and units of measure. Inconsistent reference data leads to obvious errors in downstream analytics. Symptoms include:

- Multiple representations of the same thing. For example, the state Georgia shows as "GA" and "Georgia", which makes it difficult to aggregate and drill down into data consistently.
- Difficulty aggregating data from across applications due to an inability to crosswalk the reference data values between systems. For example, the color red shows as "R" in the ERP system and "Red" in PLM system.
- Difficulty matching numbers across organizations due to differences in agreed upon reference data values for categorizing data.

### Finance 360

Financial organizations rely heavily on data for critical activities like monthly, quarterly, and annual reporting. Organizations with multiple finance and accounting systems often have financial data across multiple general ledgers, which they consolidate to produce financial reports. MDM can provide a centralized place to map and manage accounts, cost centers, business entities, and other financial data sets to a consolidated view. Symptoms include:

- Difficulty aggregating financial data across multiple systems into a consolidated view.
- Lack of process for adding and mapping new data elements in the financial systems.
- Delays in producing end of period financial reports.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

### Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

Profisee runs natively on Azure Kubernetes Service and Azure SQL Database. Both services offer out-of-the-box capabilities to support high availability.

### Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

Profisee runs natively on Azure Kubernetes Service and Azure SQL Database. You can configure Azure Kubernetes Service to scale Profisee up and out, depending on need. You can deploy Azure SQL Database in many different configurations to balance performance, scalability, and costs.

### Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Profisee authenticates users through OpenID Connect, which implements an OAuth 2.0 authentication flow. Most organizations configure Profisee to authenticate users against Microsoft Entra ID. This process ensures enterprise policies for authentication get applied and enforced.

### Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Running costs consist of a software license and Azure consumption. For more information, contact [Profisee](#).

## Deploy this scenario

To deploy this scenario:

1. Deploy Profisee into Azure using an [ARM template](#).
2. Create an [Azure Data Factory](#).
3. Configure your Azure Data Factory to [connect to a Git repository](#).
4. Add [Profisee's Azure Data Factory templates](#) to your Azure Data Factory Git repository.
5. Create a new Azure Data Factory Pipeline [using a template](#).

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Sunil Sabat](#) | Principal Program Manager

To see non-public LinkedIn profiles, sign in to LinkedIn.

## Next steps

- Understand the capabilities of the [REST Copy Connector](#) in Azure Data Factory.
- Learn more about [Profisee running natively in Azure](#).
- Learn how to deploy Profisee to Azure using an [ARM template](#).
- View the [Profisee Azure Data Factory templates](#).

## Related resources

### Architecture guides

- [Extract, transform, and load \(ETL\)](#)
- [Integration runtime in Azure Data Factory](#)
- [Choosing a data pipeline orchestration technology in Azure](#)

### Reference architectures

- [Automated enterprise BI](#)
- [Modernize mainframe & midrange data](#)
- [DataOps for the modern data warehouse](#)
- [Data warehousing and analytics](#)

# Measure Azure app sustainability by using the SCI score

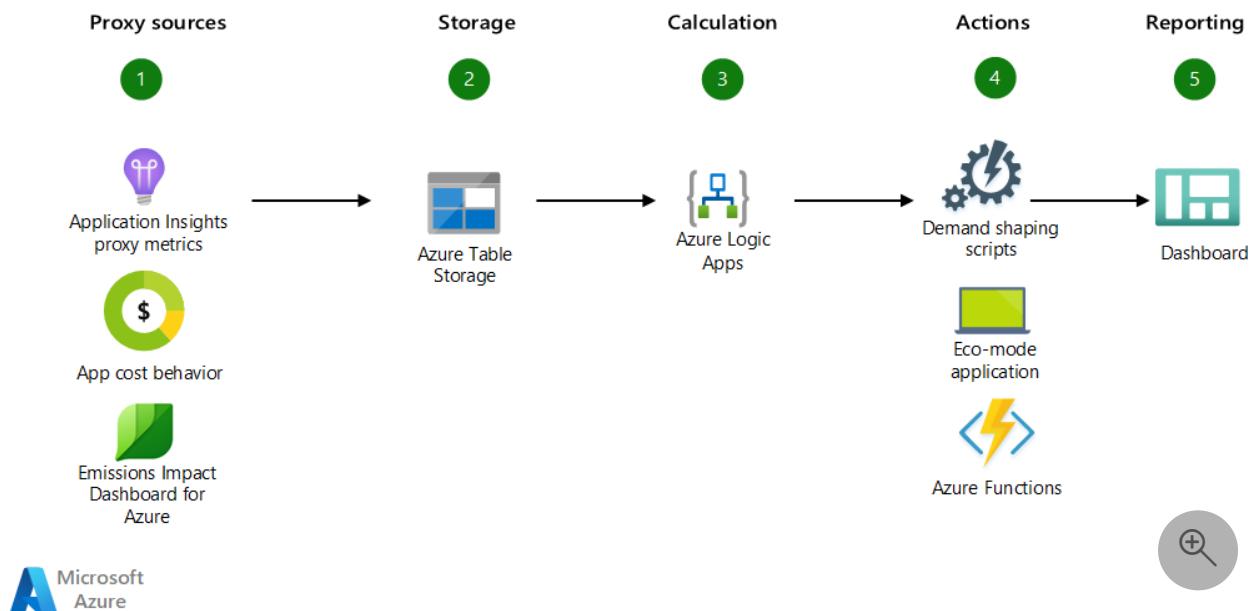
Azure Monitor   Azure Automation   Azure Logic Apps   Azure Table Storage   Power BI

This example workload helps you create a sustainability model based on available proxies. This model allows scoring of the carbon efficiency of an application. This Software Carbon Intensity (SCI) score provides a baseline for measuring changes in an application's carbon output.

## ⓘ Note

Other greenhouse gases besides carbon dioxide have different effects on the environment. For example, one ton of methane has the same heating effect as 80 tons of carbon dioxide. By convention, this article normalizes everything to the *CO2-equivalent* measure. References to *carbon* always mean the CO2-equivalent.

## Architecture



## Dataflow

1. Configure the application data sources to use to calculate the SCI score.

2. Save the data in Azure Table Storage in an Azure Storage account.
3. Use event handlers to calculate the SCI score. Event handlers might include Azure Functions, Azure Logic Apps, and Azure Blob Storage. The score is the amount of carbon emitted in grams per unit, where unit refers to the application scaling factor, or an approximation of it using proxies.
4. Use Azure Functions, Logic Apps, and automation runbooks to trigger demand shaping on the application or to initiate the pre-defined eco-mode of the application.
5. Use Power BI for reports and visualization of the score over time.

## Components

- [Emissions Impact Dashboard for Azure](#) helps measure your cloud-based emissions and carbon savings potential. It tracks direct and indirect greenhouse gas emissions related to cloud usage.
- [Application Insights](#) is an extension of [Azure Monitor](#) that provides application performance monitoring (APM). Application Insights helps you understand how people use your application. Use this knowledge to improve application efficiency.
- [Azure Table Storage](#) is a service that stores non-relational structured data, also known as *structured NoSQL data*. It provides a key/attribute store with a schemaless design. For many types of applications, access to Table Storage data is fast and cost-effective. Table Storage typically costs less than traditional SQL for similar volumes of data.
- [Azure Logic Apps](#) is a platform where you can create and run automated workflows with little to no code. By using the visual designer and selecting from prebuilt operations, build a workflow that integrates and manages proxy sources, data storage, and efficiency calculation systems.
- [Azure Functions](#) is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs. The cloud infrastructure provides all the up-to-date resources needed to keep your applications running.
- [Power BI](#) can turn data into analytics and reports that provide real-time insights. Whether your data is cloud-based or on-premises, Azure and Power BI have the integration and connectivity to bring visualizations and analytics to life.

## Alternatives

You can replace the Azure services used in this document with similar services. To do the calculation with the minimum effect on your infrastructure and to increase density and use of existing resources, use Azure services or tools that are already deployed in your environment:

- Instead of Power BI dashboards, use [Azure Monitor Workbooks](#) or [Azure Managed Grafana](#) services.
- For Application Insights, substitute another APM tool, such as [Elasticsearch](#) or Open APM.
- You can save data tables by using another system of records, such as [MySQL](#) or [MariaDB](#).
- If you have a running Azure Functions or Logic Apps applications, consider launching the calculation regularly from existing deployments.
- If the application resources are distributed across multiple resource groups, use tags to correlate cost data and calculate the amount of carbon that the application emits.

## Scenario details

These sections describe the details required to calculate a baseline for measuring changes in carbon output.

## Data sources

Try to build a proxy equation that has few variables. Choose proxy metrics that represent the application behavior and performance. This example uses the following metrics:

- The carbon emission of the infrastructure from the Emissions Impact Dashboard for Azure
- The cost of the infrastructure, measured in daily or monthly spend by resource group, from [Microsoft Cost Management](#)
- Performance and scale metrics of the application from Application Insights:
  - The number of users, API calls, or server requests that are connected to the application
  - CPU usage
  - Memory usage
  - Response time for send or receive

For a tutorial about how to set up Application Insights for the metrics, see [Application Insights SDK for ASP.NET Core applications](#).

You can add more variables to the equation, such as:

- infrastructure and edge services carbon emissions
- Time when users connect, because electricity production and demand vary with time

- Any other peculiar metric of the application that can explain how its performance changes over time

Building this equation into a score that can also reflect the number of users represents the closest approximation to a carbon score. This value is the benchmark for changes and improvements in the sustainability of the application.

Another consideration for application performance is cost. In most cases, there's a direct correlation of performance efficiency to cost and carbon savings.

[+] [Expand table](#)

Description	Conclusion
Performance is higher, but costs are the same	The application is optimized and lowered carbon emissions
Costs are lower, but performance is the same	The application is optimized and lowered carbon emissions
Performance and costs are up	The application isn't optimized and increased carbon emissions
Costs are up, but performance is lower or equal	The application isn't optimized and increased carbon emissions, or the energy cost is higher, which also causes higher carbon emissions

This correlation between application SCI score, cost, and performance is unique for every application. It depends on many factors. Gathering data for these three variables allows you to create an algorithm to forecast their variations. The SCI helps you make informed decisions about the application architecture and patterns.

## Calculations

In this scenario, process the data gathered from the Emissions Impact Dashboard as a starting point. The SCI baseline calculation is as follows:

text
$SCI = C * R$

The components are:

- `SCI`. Software Carbon Intensity result.
- `c`. The carbon emissions for the application.

This value depends on how the application is deployed in Azure. For example, if all the application resources are in a single resource group, the carbon emissions for this resource group would be the `c` variable.

 **Note**

This scenario doesn't consider other sources of emissions for the application that depend on the architecture and edge or user behavior. These considerations are the next step when you use carbon proxies.

- `R`. The scaling factor for the application.

This value can be the number of average concurrent users, for the considered time window, or API requests or web requests. The scaling factor lets the score account for the overall effect of the usage of the application, instead of just its deployment footprint.

The time window is another important aspect of this calculation. Carbon emissions vary for any energy consuming device or system, since the energy grid might have renewable or alternate energy sources at some times but not at others. For example, solar power is variable. To be as precise as possible, start with the shortest possible time frame, for example a daily or hourly calculation.

The Emissions Impact Dashboard provides monthly carbon information based on the services within a subscription. To get this number for a single resource group, use the following equation:

text

```
Carbon (res-group) = (Carbon(subscription) * Cost(res-group)) /
Cost(subscription)
```

Store the monthly carbon information for your resource group along with the rest of the data, as explained in the following section.

## Data storage

Store the carbon and carbon proxy information gathered in the previous section. Export the information to dashboards or reports, so you can visualize the carbon score over time and make informed choices. For reasons of sustainability, and in alignment with the best practices of the Well Architected Framework, use the minimum viable system of record, for example, [Azure Table Storage](#).

Tables that describe the gathered data use data like the following example:

Data from reports:

- Date
- Resource group name
- Carbon emissions from dashboard C
- Cost

Data from APM:

- CPU
- Memory
- Response time ratio (send/receive) Scaling factor R

Calculations: SCI

For more information, see:

- [Data and storage design considerations for sustainable workloads on Azure](#)
- [Application platform considerations for sustainable workloads on Azure](#).

## Data correlations

Data on the application carbon, performance, and cost allows you to build a correlation algorithm that is specific to your application. That information provides guidance when planning for cost, performance, and carbon optimization.

### Note

Equations with costs that discounts, such as Azure reservations or cost savings plans, create discrepancies in the correlation algorithm.

For more information about the choice of algorithm, see [How to select algorithms for Azure Machine Learning](#).

## Data display

You can display data and calculations several ways, such as through a customized Azure Monitor Workbook or a simple Power BI dashboard. For more information, see [Create custom KPI dashboards using Application Insights](#) and [Create a Power BI dashboard from a report](#).

## SCI score action triggers

After you score the carbon effect of an application by using proxies, the next step is to define what actions unfavorable conditions in the carbon score should trigger. Some examples of these conditions are:

- Energy production and demand are high and energy is expensive to produce
- Electricity isn't available because of natural disaster or geopolitical conflict
- Edge infrastructure becomes unavailable due to resource over-consumption or supply chain issues

After you identify the failure points that can affect the application, decide what actions to take to make the application *resilient to carbon spikes*.

Consider building an *eco-mode* version of the application. The eco-mode version is a simpler, smaller, cheaper, greener version of the full application. The application reverts to these minimal features if there are carbon emission spikes.

Consider helping end-users to choose the eco-mode version. Provide a *green button* for people to declare that they're OK with a leaner interface, fewer graphics, and limited features in exchange for reducing carbon emissions. Involving users provides an opportunity to drive cultural change along with technical change:

- Specify the effect of this choice: *By using the green version, you're saving <X> amount of carbon or bringing our carbon score to <Y>*.
- Learn about the user behavior and modify the eco-mode version to reflect their choices. For instance, if someone uses only 10 percent of application features, they might be an ideal user of the green version.
- Ideally, over time the full version is optimized for emission and the versions eventually converge.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

# Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

For more security, use [Azure Virtual Network](#) service endpoints to secure Azure service resources to only your virtual network. This approach closes public internet access to those resources and allows traffic only from your virtual network.

With this approach, you create a virtual network in Azure and then create private service endpoints for Azure services. Those services are then restricted to traffic from that virtual network. You can also reach the services from your on-premises network through a gateway.

## ⓘ Note

To move data from on-premises into Azure Storage, you need to allow public IP addresses from your on-premises computers or use [Azure ExpressRoute](#). For details, see [Deploy dedicated Azure services into virtual networks](#).

For general guidance on designing secure solutions, see the [Azure security documentation](#).

# Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

The Emissions Impact Dashboard and Azure Cost Management reports are free. This example is intentionally minimal to save on cost and carbon emissions. You can deploy this architecture by using several alternative Azure services.

Use any equivalent service you already have in your application deployment. The following resources provide component pricing information:

- [App Insights pricing](#)
- [Azure Table Storage pricing](#)
- [Azure Logic Apps pricing](#)
- [Azure Functions pricing](#)
- [Azure Automation pricing](#)
- [Power BI pricing](#)

# Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

The primary purpose of this architecture is to provide a sustainability score for your applications with a minimal effect on cost and carbon itself. Most of the components are platform as a service (PaaS) and serverless Azure services that can scale independently based on use and traffic.

The dashboard and storage interface in this example aren't suitable for heavy usage and consultation. If you plan to provide this solution to many users, consider these alternatives:

- Decouple the extracted data by transforming it and storing it in a different system of record
- Switch Azure Table Storage to a more scalable data structure alternative, such as [Azure Cosmos DB](#)

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal authors:

- [Paola Annis](#) | Principal SVC Engineering Manager
- [Jennifer Wagman](#) | Service Engineer

Other contributor:

- [Chad Kittel](#) | Principal SDE

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

This work is aligned with the principles and methodology of the [Green Software Foundation](#).

The next step to building a greener application is to embed the carbon-aware SDK into your application. You can automate triggers in real-time once you meet specific carbon

conditions. For more information, see [Green Software Foundation Carbon Aware SDK](#).

For sustainability cloud workload guidance in the Well Architected Framework, see the [Sustainability workload documentation](#).

For more information about sustainability, see these articles:

- [Build a sustainable IT infrastructure](#)
- [Reduce environmental impact of operations](#)
- [What is Microsoft Cloud for Sustainability?](#)

## Related resources

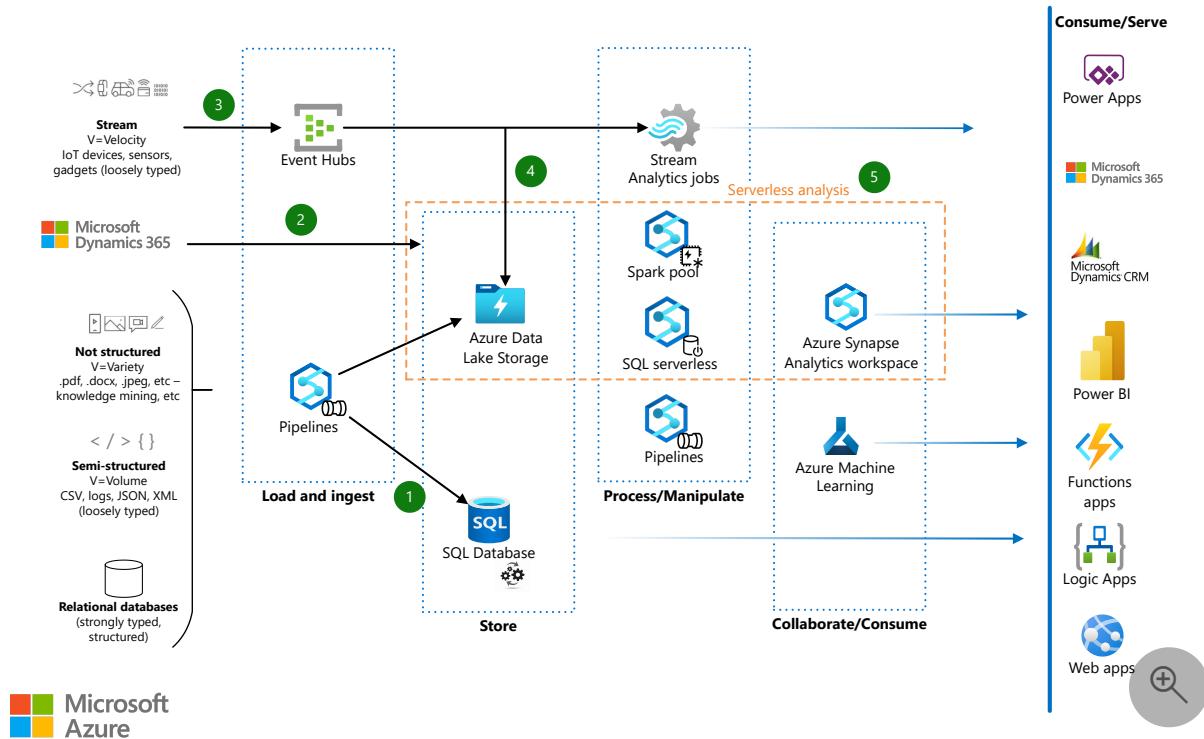
- [Choose a data analytics and reporting technology in Azure](#)
- [Data analysis workloads for regulated industries](#)
- [Interactive price analytics using transaction history data](#)
- [Power BI data write-back with Power Apps and Power Automate](#)

# Modern data warehouse for small and medium business

Azure Data Lake   Azure SQL Database   Azure Synapse Analytics   Dynamics 365   Microsoft Power Platform

This example workload shows several ways that small businesses (SMBs) can modernize legacy data stores and explore big data tools and capabilities, without overextending current budgets and skillsets. These end-to-end Azure data warehousing solutions integrate easily with tools like Azure Machine Learning, Microsoft Power Platform, Microsoft Dynamics, and other Microsoft technologies.

## Architecture



Download a [Visio file](#) of this architecture.

Legacy SMB data warehouses might contain several types of data:

- Unstructured data, like documents and graphics
- Semi-structured data, such as logs, CSVs, JSON, and XML files
- Structured relational data, including databases that use stored procedures for extract-transform-load/extract-load-transform (ETL/ELT) activities

# Dataflow

The following dataflow demonstrates the ingestion of your chosen data type:

1. Azure Synapse Analytics pipelines ingest the legacy data warehouses into Azure.
  - The pipelines orchestrate the flow of migrated or partially refactored legacy databases and SSIS packages into Azure SQL Database. This lift-and-shift approach is fastest to implement, and offers a smooth transition from an on-premises SQL solution to an eventual Azure platform-as-a-service (PaaS). You can modernize databases incrementally after the lift and shift.
  - The pipelines can also pass unstructured, semi-structured, and structured data into Azure Data Lake Storage for centralized storage and analysis with other sources. Use this approach when fusing data provides more business benefit than simply replatforming the data.
2. Microsoft Dynamics data sources can be used to build centralized BI dashboards on augmented datasets using Synapse Serverless analysis tools. You can bring the fused, processed data back into Dynamics and Power BI for further analysis.
3. Real-time data from streaming sources can also enter the system via Azure Event Hubs. For customers with real-time dashboard requirements, Azure Stream Analytics can analyze this data immediately.
4. The data can also enter the centralized Data Lake for further analysis, storage, and reporting.
5. Serverless analysis tools are available in the Azure Synapse Analytics workspace. These tools use serverless SQL pool or Apache Spark compute capabilities to process the data in Data Lake Storage Gen2. Serverless pools are available on demand, and don't require any provisioned resources.

Serverless pools are ideal for:

- Ad hoc data science explorations in T-SQL format.
- Early prototyping for data warehouse entities.
- Defining views that consumers can use, for example in Power BI, for scenarios that can tolerate performance lag.

Azure Synapse is tightly integrated with potential consumers of your fused datasets, like Azure Machine Learning. Other consumers can include Power Apps, Azure Logic Apps, Azure Functions apps, and Azure App Service web apps.

# Components

- [Azure Synapse Analytics](#) is an analytics service that combines data integration, enterprise data warehousing, and big data analytics. In this solution:
  - An [Azure Synapse Workspace](#) promotes collaboration between data engineers, data scientists, data analysts, and business intelligence (BI) professionals.
  - [Azure Synapse pipelines](#) orchestrate and ingest data into SQL Database and Data Lake Storage Gen2.
  - [Azure Synapse serverless SQL pools](#) analyze unstructured and semi-structured data in Data Lake Storage Gen2 on demand.
  - [Azure Synapse serverless Apache Spark pools](#) do code-first explorations in Data Lake Storage Gen2 with Spark languages like Spark SQL, pySpark, and Scala.
- [Azure SQL Database](#) is an intelligent, scalable, relational database service built for the cloud. In this solution, SQL Database holds the enterprise data warehouse and performs ETL/ELT activities that use stored procedures.
- [Azure Event Hubs](#) is a real-time data streaming platform and event ingestion service. Event Hubs can ingest data from anywhere, and seamlessly integrates with Azure data services.
- [Azure Stream Analytics](#) is a real-time, serverless analytics service for streaming data. Stream Analytics offers rapid, elastic scalability, enterprise-grade reliability and recovery, and built-in machine learning capabilities.
- [Azure Machine Learning](#) is a toolset for data science model development and lifecycle management. Machine Learning is one example of the Azure and Microsoft services that can consume fused, processed data from Data Lake Storage Gen2.

# Alternatives

- [Azure IoT Hub](#) could replace or complement Event Hubs. The solution you choose depends on the source of your streaming data, and whether you need cloning and bidirectional communication with the reporting devices.
- You can use [Azure Data Factory](#) for data integration instead of Azure Synapse pipelines. The choice depends on several factors:
  - Azure Synapse pipelines keep the solution design simpler, and allow collaboration inside a single Azure Synapse workspace.
  - Azure Synapse pipelines don't support SSIS packages rehosting, which is available in Azure Data Factory.

- [Synapse Monitor Hub](#) monitors Azure Synapse pipelines, while [Azure Monitor](#) can monitor Data Factory.

For more information and a feature comparison between Azure Synapse pipelines and Data Factory, see [Data integration in Azure Synapse Analytics versus Azure Data Factory](#).

- You can use [Synapse Analytics dedicated SQL pools](#) for storing enterprise data, instead of using SQL Database. Review the use cases and considerations in this article and related resources to make a decision.

## Scenario details

Small and medium businesses (SMBs) face a choice when modernizing their on-premises data warehouses for the cloud. They can adopt big data tools for future extensibility, or keep traditional, SQL-based solutions for cost efficiency, ease of maintenance, and smooth transition.

However, a hybrid approach combines easy migration of the existing data estate with the opportunity to add big data tools and processes for some use cases. SQL-based data sources can keep running in the cloud and continue to modernize as appropriate.

This example workload shows several ways that SMBs can modernize legacy data stores and explore big data tools and capabilities, without overextending current budgets and skillsets. These end-to-end Azure data warehousing solutions integrate easily with Azure and Microsoft services and tools like Azure Machine Learning, Microsoft Power Platform, and Microsoft Dynamics.

## Potential use cases

Several scenarios can benefit from this workload:

- Migrating a traditional, on-premises relational data warehouse that's smaller than 1 TB and extensively uses SQL Server Integration Services (SSIS) packages to orchestrate stored procedures.
- Meshing existing Dynamics or Power Platform [Dataverse](#) data with batched and real-time [Azure Data Lake](#) sources.
- Using innovative techniques to interact with centralized Data Lake Storage Gen2 data. Techniques include serverless analysis, knowledge mining, data fusion between domains, and end-user data exploration.

- Setting up eCommerce companies to adopt a data warehouse to optimize their operations.

This solution isn't recommended for:

- [Greenfield](#) deployment of data warehouses that are estimated to be > 1 TB within one year.
- Migrating on-premises data warehouses that are > 1 TB or projected to grow to that size within a year.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

The following considerations apply to this scenario.

## Availability

SQL Database is a PaaS service that can meet your high availability (HA) and disaster recovery (DR) requirements. Be sure to pick the SKU that meets your requirements. For guidance, see [High availability for Azure SQL Database](#).

## Operations

SQL Database uses [SQL Server Management Studio \(SSMS\)](#) to develop and maintain legacy artifacts like stored procedures.

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

See a [pricing sample for a SMB data warehousing scenario](#) in the Azure pricing calculator. Adjust the values to see how your requirements affect the costs.

- [SQL Database](#) bases costs on the selected Compute and Service tiers, and the number of vCores and Database Transaction Units (DTUs). The example shows a

single database with provisioned Compute and eight vCores, based on the assumption that you need to run stored procedures in SQL Database.

- [Data Lake Storage Gen2](#) pricing depends on the amount of data you store and how often you use the data. The sample pricing includes 1 TB of data stored, with further transactional assumptions. The 1 TB refers to the size of the data lake, not the original legacy database size.
- [Azure Synapse pipelines](#) base costs on the number of data pipeline activities, integration runtime hours, data flow cluster size, and execution and operation charges. Pipeline costs increase with additional data sources and amounts of data processed. The example assumes one data source batched every hour for 15 minutes on an Azure-hosted integration runtime.
- [Azure Synapse Spark pool](#) bases pricing on node size, number of instances, and uptime. The example assumes one small compute node with five hours a week to 40 hours a month utilization.
- [Azure Synapse serverless SQL pool](#) bases pricing on TBs of data processed. The sample assumes 50 TBs processed a month. This figure refers to the size of the data lake, not the original legacy database size.
- [Event Hubs](#) bills based on tier, throughput units provisioned, and ingress traffic received. The example assumes one throughput unit in Standard tier over one million events for a month.
- [Stream Analytics](#) bases costs on the number of provisioned streaming units. The sample assumes one streaming unit used over the month.

## Contributors

*This article is being updated and maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- Galina Polyakova | Senior Cloud Solution Architect

## Next steps

- For training content and labs, see the [Data Engineer Learning Paths](#).
- [Tutorial: Get started with Azure Synapse Analytics](#)
- [Create a single database - Azure SQL Database](#)

- [Create a storage account for Azure Data Lake Storage Gen2](#)
- [Azure Event Hubs Quickstart - Create an event hub using the Azure portal](#)
- [Quickstart - Create a Stream Analytics job by using the Azure portal](#)
- [Quickstart: Get started with Azure Machine Learning](#)

## Related resources

- Learn more about:
  - [Data lakes](#)
  - [Data warehousing and analytics](#)
  - [Analytics end-to-end with Azure Synapse](#)
  - [Big data analytics with enterprise-grade security using Azure Synapse](#)
  - [Enterprise business intelligence](#)

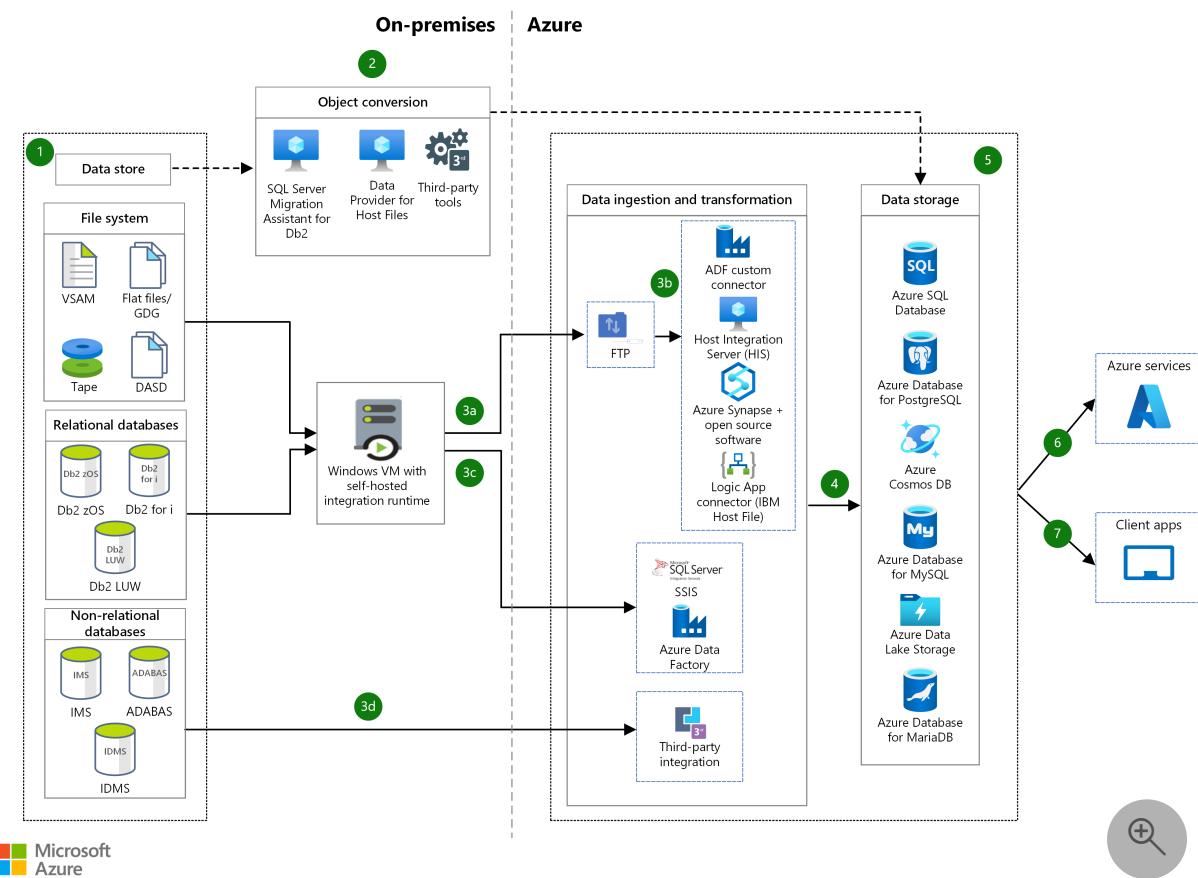
# Modernize mainframe and midrange data

Azure Cosmos DB   Azure Data Lake   Azure SQL Database   Azure SQL Managed Instance   Azure Storage

Apache®, [Spark](#), and the flame logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

This article describes an end-to-end modernization plan for mainframe and midrange data sources.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

The following dataflow outlines a process for modernizing a mainframe data tier. It corresponds to the preceding diagram.

1. Mainframe and midrange systems store data in data sources, like file systems (VSAM, flat file, LTFS), relational databases (Db2 for z/OS, Db2 for IBM i, Db2 for Linux UNIX and Windows), or non-relational databases (IMS, ADABAS, IDMS).
2. The object conversion process extracts object definitions from source objects. The definitions are then converted into corresponding objects in the target data store.
  - [SQL Server Migration Assistant](#) (SSMA) for Db2 migrates schemas and data from IBM Db2 databases to Azure databases.
  - [Managed Data Provider for Host Files](#) converts objects by:
    - Parsing COBOL and RPG record layouts, or *copybooks*.
    - Mapping the copybooks to C# objects that .NET applications use.
  - Third-party tools perform automated object conversion on non-relational databases, file systems, and other data stores.
3. Data is ingested and transformed. Mainframe and midrange systems store their file system data in EBCDIC-encoded format in file formats like:
  - Indexed [VSAM](#) files
  - Non-indexed [GDG](#) files
  - Flat files

COBOL, PL/I, and assembly language copybooks define the data structure of these files.

- a. FTP transfers mainframe and midrange file system datasets with single layouts and unpacked fields in binary format and corresponding copybook to Azure.
- b. Data is converted. The Azure Data Factory custom connector is a solution developed by using the Host File client component of Host Integration Server to convert mainframe datasets.

[Host Integration Server](#) integrates existing IBM host systems, programs, messages, and data with Azure applications. Host Integration Server is a Host File client component that you can use to develop a custom solution for dataset conversion.

The Azure Data Factory custom connector is based on the open-source Spark framework, and it runs on [Azure Synapse Analytics](#). Like other solutions, it can parse the copybook and convert data. Manage the service for data conversion by using the [Azure Logic Apps](#) Parse Host File Contents connector.

- c. Relational database data is migrated.

IBM mainframe and midrange systems store data in relational databases like these:

- [Db2 for z/OS ↗](#)
- [Db2 for Linux UNIX and Windows ↗](#)
- [Db2 for IBM i ↗](#)

These services migrate the database data:

- Data Factory uses a Db2 connector to extract and integrate data from the databases.
- SQL Server Integration Services handles various data [ETL ↗](#) tasks.

d. Non-relational database data is migrated.

IBM mainframe and midrange systems store data in non-relational databases like these:

- [IDMS ↗](#), a [network model ↗](#) database management system (DBMS)
- [IMS ↗](#), a [hierarchical model ↗](#) DBMS
- [Adabas ↗](#)
- [Datacom ↗](#)

Third-party products integrate data from these databases.

4. Azure services like Data Factory and [AzCopy](#) load data into Azure databases and Azure data storage. You can also use third-party solutions and custom loading solutions to load data.

5. Azure provides many managed data storage solutions:

- Databases:
  - [Azure SQL Database](#)
  - [Azure Database for PostgreSQL](#)
  - [Azure Cosmos DB](#)
  - [Azure Database for MySQL](#)
  - [Azure Database for MariaDB](#)
  - [Azure SQL Managed Instance](#)
- Storage:
  - [Azure Data Lake Storage](#)
  - [Azure Blob Storage](#)

6. Azure services use the modernized data tier for computing, analytics, storage, and networking.

7. Client applications also use the modernized data tier.

# Components

## Data storage

- [SQL Database](#) is part of the [Azure SQL family](#). It's built for the cloud and provides all the benefits of a fully managed and evergreen platform as a service. SQL Database also provides AI-powered automated features that optimize performance and durability. Serverless compute and [Hyperscale storage options](#) automatically scale resources on demand.
- [Azure Database for PostgreSQL](#) is a fully managed relational database service that's based on the community edition of the open-source [PostgreSQL](#) database engine.
- [Azure Cosmos DB](#) is a globally distributed [multimodel](#) [NoSQL](#) database.
- [Azure Database for MySQL](#) is a fully managed relational database service that's based on the community edition of the open-source [MySQL](#) database engine.
- [Azure Database for MariaDB](#) is a cloud-based relational database service. It's based on the [MariaDB](#) community edition database engine.
- [SQL Managed Instance](#) is an intelligent, scalable cloud database service that offers all the benefits of a fully managed and evergreen platform as a service. SQL Managed Instance has near-100% compatibility with the latest SQL Server Enterprise edition database engine. It also provides a native virtual network implementation that addresses common security concerns.
- [Azure Data Lake Storage](#) is a storage repository that holds large amounts of data in its native, raw format. Data lake stores are optimized for scaling to terabytes and petabytes of data. The data typically comes from multiple heterogeneous sources. It can be structured, semi-structured, or unstructured.

## Compute

- Data Factory integrates data across different network environments by using an [integration runtime](#) (IR), which is a compute infrastructure. Data Factory copies data between cloud data stores and data stores in on-premises networks by using [self-hosted IRs](#).
- [Azure Virtual Machines](#) provides on-demand, scalable computing resources. An Azure virtual machine (VM) provides the flexibility of virtualization but eliminates the maintenance demands of physical hardware. Azure VMs offer a choice of operating systems, including Windows and Linux.

## Data integrators

- [Azure Data Factory](#) is a hybrid data integration service. In this solution, an Azure Data Factory custom connector uses the Host File client component of Host Integration Server to convert mainframe datasets. With minimal setup, you can use a custom connector to convert your mainframe dataset just as you'd use any other Azure Data Factory connector.
- [AzCopy](#) is a command-line utility that moves blobs or files into and out of storage accounts.
- [SQL Server Integration Services](#) is a platform for creating enterprise-level data integration and transformation solutions. You can use it to solve complex business problems by:
  - Copying or downloading files.
  - Loading data warehouses.
  - Cleansing and mining data.
  - Managing SQL Server objects and data.
- [Host Integration Server](#) technologies and tools enable you to integrate existing IBM host systems, programs, messages, and data with Azure applications. The Host File client component provides flexibility for data that's converted from EBCDIC to ASCII. For example, you can generate JSON/XML from the data that's converted.
- [Azure Synapse](#) brings together data integration, enterprise data warehousing, and big data analytics. The Azure Synapse conversion solution used in this architecture is based on Apache Spark and is a good candidate for large mainframe-dataset workload conversion. It supports a wide range of mainframe data structures and targets and requires minimal coding effort.

## Other tools

- [SQL Server Migration Assistant for Db2](#) automates migration from Db2 to Microsoft database services. When it runs on a VM, this tool converts Db2 database objects into SQL Server database objects and creates those objects in SQL Server.
- [Data Provider for Host Files](#) is a component of [Host Integration Server](#) that uses offline, SNA, or TCP/IP connections.
  - With offline connections, Data Provider reads and writes records in a local binary file.
  - With SNA and TCP/IP connections, Data Provider reads and writes records stored in remote z/OS (IBM Z Series Mainframe) datasets or remote i5/OS (IBM AS/400 and iSeries systems) physical files. Only i5/OS systems use TCP/IP.
- [Azure services](#) provide environments, tools, and processes for developing and scaling new applications in the public cloud.

# Scenario details

Modern data storage solutions like the Azure data platform provide better scalability and performance than mainframe and midrange systems. By modernizing your systems, you can take advantage of these benefits. However, updating technology, infrastructure, and practices is complex. The process involves an exhaustive investigation of business and engineering activities. Data management is one consideration when you modernize your systems. You also need to look at data visualization and integration.

Successful modernizations use a [data-first strategy](#). When you use this approach, you focus on the data rather than the new system. Data management is no longer just an item on the modernization checklist. Instead, the data is the centerpiece. Coordinated, quality-oriented data solutions replace fragmented, poorly governed ones.

This solution uses Azure data platform components in a data-first approach. Specifically, the solution involves:

- **Object conversion.** Converting object definitions from the source data store to corresponding objects in the target data store.
- **Data ingestion.** Connecting to the source data store and extracting data.
- **Data transformation.** Transforming extracted data into appropriate target data store structures.
- **Data storage.** Loading data from the source data store to the target data store, both initially and continually.

## Potential use cases

Organizations that use mainframe and midrange systems can benefit from this solution, especially when they want to achieve these goals:

- Modernize mission-critical workloads.
- Acquire business intelligence to improve operations and gain a competitive advantage.
- Remove the high costs and rigidity that are associated with mainframe and midrange data stores.

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, a set of guiding tenets that you can use to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#). When you use the Data

Provider for Host Files client to convert data, [turn on connection pooling](#) to reduce the connection startup time. When you use Data Factory to extract data, [tune the performance of the copy activity](#).

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

- Be aware of the differences between on-premises client identities and client identities in Azure. You need to compensate for any differences.
- Use [managed identities](#) for component-to-component data flows.
- When you use Data Provider for Host Files to convert data, follow the recommendations in [Data Providers for Host Files security and protection](#).

## Cost optimization

Cost optimization is about reducing unnecessary expenses and improving operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

- SQL Server Migration Assistant is a free, supported tool that simplifies database migration from Db2 to SQL Server, SQL Database, and SQL Managed Instance. SQL Server Migration Assistant automates all aspects of migration, including migration assessment analysis, schema and SQL statement conversion, and data migration.
- The Azure Synapse Spark-based solution is built from open-source libraries. It eliminates the financial burden of licensing conversion tools.
- Use the [Azure pricing calculator](#) to estimate the cost of implementing this solution.

## Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see the [Performance efficiency pillar overview](#).

- The key pillars of performance efficiency are performance management, capacity planning, [scalability](#), and choosing an appropriate performance pattern.
- You can [scale out the self-hosted IR](#) by associating the logical instance with multiple on-premises machines in active-active mode.
- Azure SQL Database offers the ability to dynamically scale your databases. In a serverless tier, it can automatically scale the compute resources. Elastic Pool, which

allows databases to share resources in a pool, can only be scaled manually.

# Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Ashish Khandelwal](#) | Principal Engineering Architect Manager

Other contributors:

- [Mick Alberts](#) | Technical Writer
- [Nithish Aruldoss](#) | Engineering Architect

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

Review the [Azure Database Migration Guides](#). Contact [Azure Data Engineering - Mainframe & Midrange Modernization](#) for more information.

See these articles:

- [IBM workloads on Azure](#)
- [Mainframe rehosting on Azure virtual machines](#)
- [Mainframe workloads supported on Azure](#)
- [Move mainframe compute to Azure](#)

## Related resources

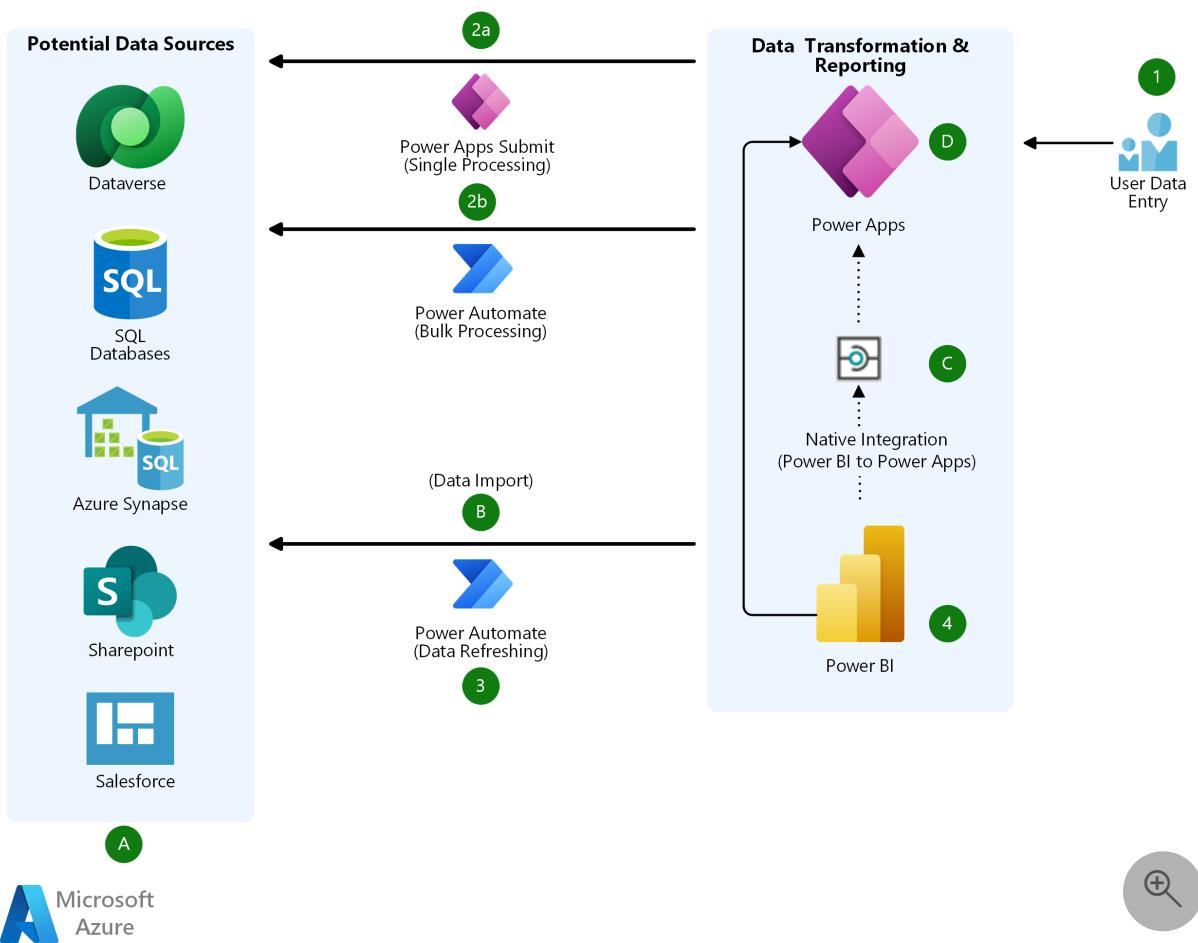
- [Azure data platform end-to-end](#)

# Power BI data write-back with Power Apps and Power Automate

Power BI Power Apps Power Automate Microsoft Power Platform

This solution implements a Power Apps canvas app via the native Power Apps visual in Power BI. Power Automate provides background automation for bulk-processing and refreshing.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

Core components of this solution incorporate the ability to pass pre-filtered data from Power BI into a Power Apps and/or Power Automate funnel for any updates in a

supporting back end. It's important to refresh the Power BI data set (or dataflow) to ensure updates are visible to all users.

## Deployment

(For more information, see [Deploy this scenario](#) later in this article.)

- A. Deploy Dataverse and a supporting model-driven app with relevant custom tables.
- B. Import all back-end tables and views into a Power BI data set (PBIX).
- C. Initiate the integration between Power BI and Power Apps via the Power Apps visualization in the desktop application.
- D. Use Power Apps to create a canvas app to provide the ability to interact with and update all necessary data.

## Process flow

1. **Gather data.** Cross filter a selected row or set of data by selecting part of a visualization in a Power BI report. This interaction passes the necessary underlying data from the Power BI report interface into the embedded canvas app.
2. **Update data or insert it into Dataverse by using the UI of the canvas app.** You can do that by using bound controls like forms and galleries that are native to Power Apps and that are directly tied to back-end data. Alternatively, you can implement more customized functionality by using unbound controls. These controls require additional Power Fx code. For single-update scenarios, you can code the app to directly commit data to the back end via **SubmitForm**, **Patch**, and **UpdateIf** functions. For bulk-update scenarios, you can establish a collection (a virtual table) by using the **Collect** function. You can then pass the collection to process all data updates at once. See [Power App UI](#) for screenshots of the canvas app.
3. **Push updates to the source.** A Power Automate flow provides background automation as required by the scenario. For single-update scenarios where only one row from the selected table is updated, a simple flow runs to refresh the PBIX data set. This ensures that the updated data is reflected in the Dataverse back end and in the reporting layer. For bulk-update scenarios, a more complex flow runs. It consumes a JSON collection of nested objects that's passed from the Power Apps collection described in the previous step. The flow then iterates through each nested object, individually updating data in Dataverse as needed. After the update step completes, the flow refreshes the PBIX. If the Power BI report uses

DirectQuery, the automated steps associated with refreshing the PBIX aren't needed.

4. **Visualize updates.** All data is updated and refreshed. The end user refreshes the browser window to see the update.

## Components

- **Dataverse.** [↗](#) A back-end database solution that you can use to store data in a highly secure, customizable, scalable environment. This environment seamlessly connects to Dynamics 365, Azure, Visual Studio, and Power Query. Dataverse provides efficient data processing and an open-source shared data model that provides semantic consistency.
- **Power BI.** [↗](#) A collection of software services, apps, and connectors that work together to turn your unrelated sources of data into coherent, visually immersive, interactive insights.
  - You can also implement data write-back directly into [Power Query dataflows](#).
- **Power Apps.** [↗](#) A suite of apps, services, and connectors, all available on a comprehensive data platform. You can use this service to quickly create applications to meet custom business needs. In this solution, Power Apps is used for data updates and inserts in an intuitive UI. It also functions as a trigger for automation.
- **Power Automate.** [↗](#) A service that you can use to create automated workflows between a variety of connected apps and outside services. You can configure it to transfer data, send notifications, collect artifacts, and more. In this solution, Power Automate is used for bulk-processing of updated data and for data refresh in the PBIX and/or Dataflow layer to push updated data back into a Power BI report.

## Alternatives

- Alternatives to Dataverse include the following solutions:
  - [Azure SQL Database](#) [↗](#)
  - [Azure Synapse Analytics](#) [↗](#)
  - [SQL Server](#) [↗](#)
  - [Salesforce](#) [↗](#)
  - [SharePoint](#) [↗](#)
- You can use [Power Query dataflows](#) separately or together with Power BI data sets for this solution, depending on the scale and efficiency of data in your environment. If you use dataflows in your solution, you need to manage your Power Automate extension to refresh each dataflow or data set accordingly.

- You can build custom applications by using JavaScript, HTML, C#, or other languages that can be embedded into a Power BI report to update selected data. These apps, however, need to be implemented differently in the Power BI report layer because there's no native visualization as there is for Power Apps. If you implement scalability for these apps, you need to monitor it. For information on how to best implement custom components in Power BI, see the [Power BI Developer Center](#).
- You can also use the [Power Automate visual for Power BI](#) for write-back scenarios. This visual is optimized for handling large sets of data, and Power Apps handles delegation. You can use the Power Automate and Power Apps visuals together to provide scalable efficiency. If you use the Power Automate visual, data update occurs in the background without the presence of a displayed UI.

## Scenario details

This solution for data write-back functionality in Power BI provides an interactive and efficient way to change data directly from Power BI. Power BI doesn't currently have a native solution that you can use for inline or bulk updates of data while you're interacting with a report or dashboard. To push changes to data, you need to make updates directly in your data stores and then, if you're not using DirectQuery, refresh a data set to complete the process flow. This process can be inefficient and can pose problems for users who don't have access to a specific back end or the underlying data.

## Potential use cases

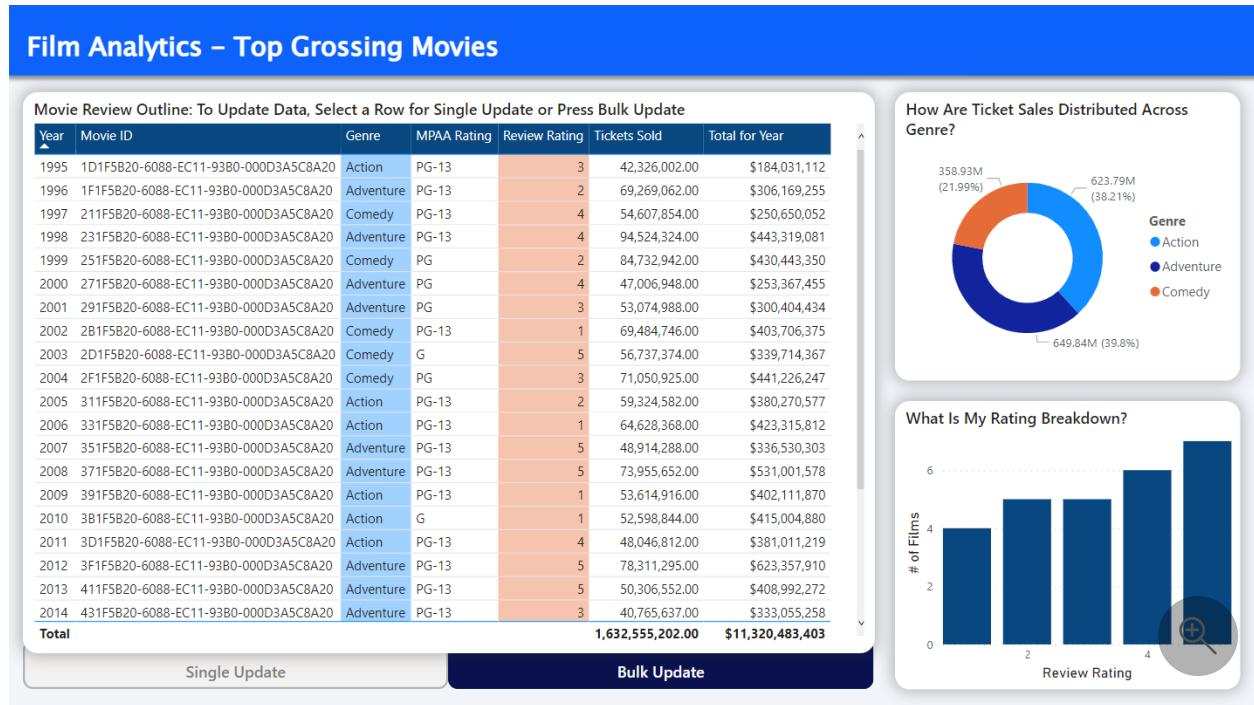
This architecture is highly iterative. You can use it with several different back-end data stores and adapt it to various use cases. Practical uses for this architecture include:

- **Inline editing.** The solution can be used for data that needs to be updated on the fly without provisioned access to a back-end database.
- **Approval workflows.** Extending the capabilities of Power BI with Power Apps and Power Automate allows end users to collect data that requires review directly from a dashboard and send it to subsequent approvers.
- **Data-driven alerts.** The solution can provide customization to automated notifications about specific insights via submission of records or the passing of data packets into a Power Automate flow.

## Power App UI

The following screenshots illustrate the process for passing data from Power BI to the underlying database.

This is the home screen for the canvas app:



This screenshot shows the process for a single update:

Would You Like To Update A Row?:  
Select a Row Below to Use the Power App on the Right

Year	Movie ID	Genre	Review Rating
1995	1D1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	3

**Data Write-Back**

Single Update      Bulk Update

Year  
1995

Name  
1d1f5b20-6088-ec11-93b0-000d3a5c8a20

Genre  
Action

Review Rating

Submit

This screenshot shows the process for a bulk update:

Year	Movie ID	Genre	Review Rating
1995	1D1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	3
1996	1F1F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	2
1997	211F5B20-6088-EC11-93B0-000D3A5C8A20	Comedy	4
1998	231F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	4
1999	251F5B20-6088-EC11-93B0-000D3A5C8A20	Comedy	2
2000	271F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	4
2001	291F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	3
2002	2B1F5B20-6088-EC11-93B0-000D3A5C8A20	Comedy	1
2003	2D1F5B20-6088-EC11-93B0-000D3A5C8A20	Comedy	5
2004	2F1F5B20-6088-EC11-93B0-000D3A5C8A20	Comedy	3
2005	311F5B20-6088-EC11-93B0-000D3A5C8A20	Action	2
2006	331F5B20-6088-EC11-93B0-000D3A5C8A20	Action	1
2007	351F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	5
2008	371F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	5
2009	391F5B20-6088-EC11-93B0-000D3A5C8A20	Action	1
2010	3B1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	1
2011	3D1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	4
2012	3F1F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	5
2013	411F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	5
2014	431F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	3
2015	451F5B20-6088-EC11-93B0-000D3A5C8A20	Action	3
2016	471F5B20-6088-EC11-93B0-000D3A5C8A20	Action	5
2017	491F5B20-6088-EC11-93B0-000D3A5C8A20	Action	2
2018	4B1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	4
2019	4D1F5B20-6088-EC11-93B0-000D3A5C8A20	Adventure	5
2020	4F1F5B20-6088-EC11-93B0-000D3A5C8A20	Action	2

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, a set of guiding tenets that you can use to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

## Scalability

To correctly establish the integration between Power BI and the canvas app for write-back support, you need to set it up by creating the canvas app directly from the Power Apps visualization on the Power BI report. If this integration isn't set up correctly, there will be no way to pass cross-filtered data from the Power BI report layer to the Power Apps UI.

You need to address [delegation](#) when you consider scalability. Delegation is a concept that's unique to Power Apps (canvas apps) that limits the scope of data processing through the cloud while an app's logic is running. Canvas apps implemented in this solution need to be provisioned properly to handle large sets of data that use loops or complex filter statements to ensure that all data is covered when you run an update to the back-end database and then the Power BI data set. You can use Power Automate in

this scenario to increase efficiency when you handle large-scale bulk updates of more than 2,000 rows.

## Availability

All the components outlined in this architecture are managed services that automatically scale depending on regional availability. Currently, Power Apps is available in six core regions and 42 languages. For more information, see [availability of services](#).

**Dataverse** is designed to meet enterprise-level scalability needs by using service-protection limits to mitigate malicious behavior that might disrupt service.

For information about SLAs, see [Service-level agreements](#).

## Security

**Row-level security (RLS)** is the best way to restrict data access for individual users or groups in Power BI. RLS models persist in this solution. If a user's permissions in Power BI are set to view only a subset of the overall data model, only that subset can be passed to the Power Apps layer. However, you need to configure the Power Apps layer so that end users are able to access only certain data.

You configure data security for Power Apps by using [role-based security](#) in the Dataverse back end. You can apply roles to teams, groups, or individual users to specify which records are available for manipulation in this solution. This functionality enables you to use a single canvas app for users who have different levels of access to the back end. To ensure consistency across the solution, be sure the role-based security configurations match the permissions outlined in the Power BI row-level security model for each team, group, or user.

For more information on how to implement a well-architected framework, see the [Microsoft security pillar](#).

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

**Power Apps** and **Power Automate** are software as a service (SaaS) applications that have flexible pricing models. You can license Power Apps with per-app or per-user plans

that fit your business needs. Similarly, you can license Power Automate with either per-user or per-flow (single automation) plans.

Several versions of [Power BI](#) are available. Your choice depends on the volume of data ingested. For Power BI Pro, only a per-user plan is available. Power BI Premium provides per-user and per-capacity plans.

## Deploy this scenario

You need to have the relevant Power Platform licenses to run this solution in production. Administrators or customizers of the solution also need the proper security roles to enable access to Power Apps and Power Automate. If you don't yet have access to these licenses or roles, you can use the [Power Apps Developer Plan](#) to start development in the meantime.

To deploy this solution:

1. Create a PBIX file in Power BI Desktop as the base component of your reporting layer. Import all necessary data from Dataverse or whichever back end you're using.
2. Add the Power Apps visual for Power BI directly from the **Visualization** pane. Creating an app directly from the Power Apps visual for Power BI is the only way to implement integration between Power BI and Power Apps.
3. After you implement the integration, you need to develop, design, and code the canvas app to perform the business processes you want to streamline.
4. If bulk processing is required, a developer needs to create a Power Automate flow to handle the consumption of data from Power Apps and its propagation to Dataverse. You can configure this flow to provide any notifications or approval workflows that you want to incorporate in the automation.
5. When the app is complete, you need to incorporate it into the Power BI report. You can do this directly in the Power BI report screen or by configuring a drillthrough page.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal authors:

- [Tom Berzok](#) | Consultant, Data & Analytics at [Slalom](#)
- [Thomas Edmondson](#) | Principal at [Slalom](#)

Other contributor:

- [Mick Alberts](#) | Technical Writer

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

Product documentation:

- [Power Apps visual for Power BI](#)
- [Overview of creating apps in Power Apps](#)
- [Embed a Power Apps visual in a Power BI report](#)
- [Use a flow to update a row in Dataverse](#)

Microsoft Learn Training modules:

- [Create tables in Dataverse](#)
- [Get started with Power Apps canvas apps](#)
- [Manage solutions in Power Apps and Power Automate](#)

## Related resources

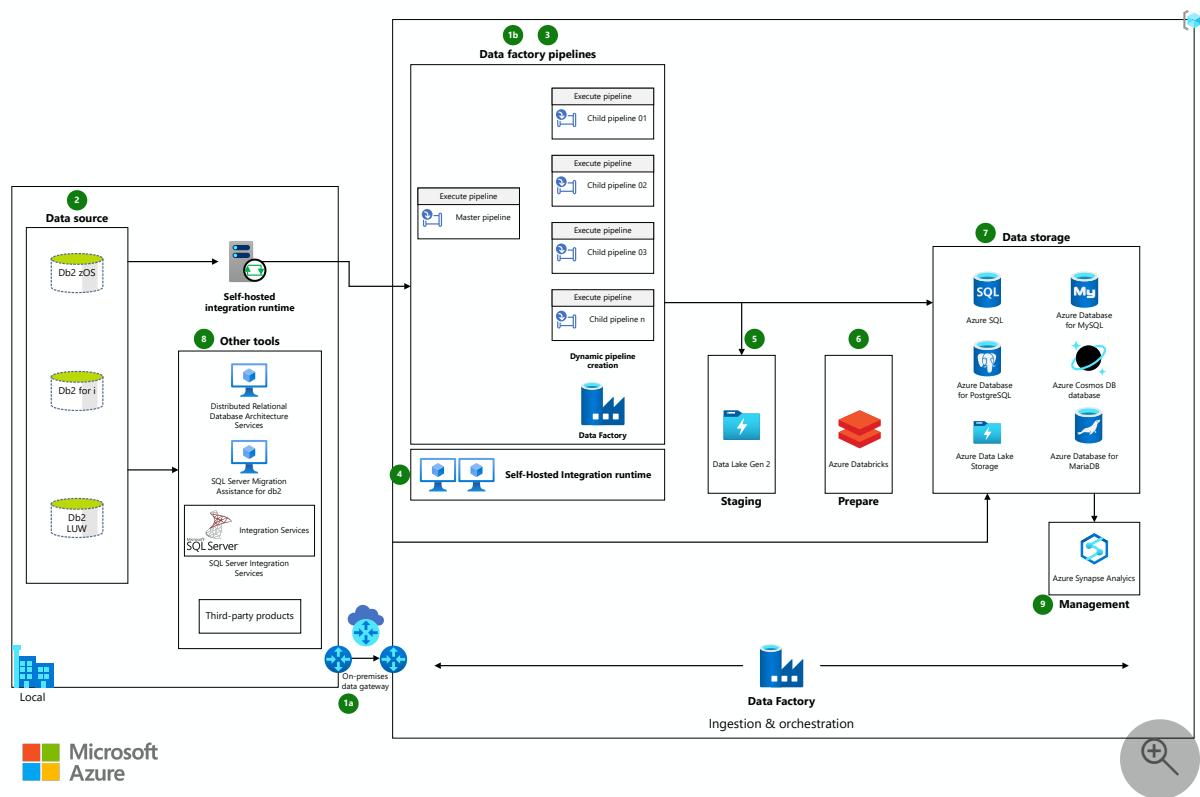
- [Advanced analytics architecture](#)
- [Extract text from objects using Power Automate and AI Builder](#)
- [Custom business processes](#)
- [Data governance with Profisee and Azure Purview](#)

# Replicate and sync mainframe data in Azure

Azure Data Factory    Azure Databricks

This reference architecture outlines an implementation plan for replicating and syncing data during modernization to Azure. It discusses technical aspects like data stores, tools, and services.

## Architecture



Download a [Visio file](#) of this architecture.

## Workflow

Mainframe and midrange systems update on-premises application databases on a regular interval. To maintain consistency, the solution syncs the latest data with Azure databases. The sync process involves the following steps:

1. These actions occur throughout the process:

- a. An on-premises data gateway transfers data quickly and securely between on-premises systems and Azure services. With this configuration, the on-premises data gateway can receive instructions from Azure and replicate data without the on-premises network directly exposing the local data assets.
  - b. Azure Data Factory pipelines orchestrate activities that range from data extraction to data loading. You can schedule pipeline activities, start them manually, or automatically trigger them.
2. On-premises databases like Db2 zOS, Db2 for i, and Db2 LUW store the data.
  3. Pipelines group the activities that perform tasks. To extract data, Data Factory dynamically creates one pipeline per on-premises table. You can then use a massively parallel implementation when you replicate data in Azure. But you can also configure the solution to meet your requirements:
    - Full replication: You replicate the entire database, making necessary modifications to data types and fields in the target Azure database.
    - Partial, delta, or incremental replication: You use *watermark columns* in source tables to sync updated rows with Azure databases. These columns contain either a continuously incrementing key or a time stamp indicating the table's last update.
- Data Factory also uses pipelines for the following transformation tasks:
- Data type conversion
  - Data manipulation
  - Data formatting
  - Column derivation
  - Data flattening
  - Data sorting
  - Data filtering
4. A self-hosted integration runtime (IR) provides the environment that Data Factory uses to run and dispatch activities.
  5. Azure Data Lake Storage Gen2 and Azure Blob Storage provide a place for data staging. This step is sometimes required for transforming and merging data from multiple sources.
  6. Data preparation takes place next. Data Factory uses Azure Databricks, custom activities, and pipeline data flows to transform data quickly and effectively.
  7. Data Factory loads data into relational and non-relational Azure databases:

- Azure SQL
- Azure Database for PostgreSQL
- Azure Cosmos DB
- Azure Data Lake Storage
- Azure Database for MariaDB
- Azure Database for MySQL

In certain use cases, other tools can also load data.

## 8. Other tools can also replicate and transform data:

- Microsoft Service for Distributed Relational Database Architecture (DRDA): These DRDA services can connect to the Azure SQL family of databases and keep on-premises databases up to date. These services run on an on-premises virtual machine (VM) or an Azure VM.
- SQL Server Migration Assistance (SSMA) for Db2: This tool migrates schemas and data from IBM Db2 databases to Azure databases.
- SQL Server Integration Services (SSIS): This platform can extract, transform, and load data.
- Third-party tools: When the solution requires near real-time replication, you can use third-party tools. Some of these agents are available in [Azure Marketplace](#).

## 9. Azure Synapse Analytics manages the data and makes it available for business intelligence and machine learning applications.

# Components

The solution uses the following components:

## Tools

- [Microsoft Service for DRDA](#) is a component of [Host Integration Server \(HIS\)](#). Microsoft Service for DRDA is an Application Server (AS) that DRDA Application Requester (AR) clients use. Examples of DRDA AR clients include IBM Db2 for z/OS and Db2 for i5/OS. These clients use the AS to convert Db2 SQL statements and run them on SQL Server.
- [SSMA for Db2](#) automates migration from Db2 to Microsoft database services. While running on a VM, this tool converts Db2 database objects into SQL Server database objects and creates those objects in SQL Server. SSMA for Db2 then migrates data from Db2 to the following services:

- SQL Server 2012
  - SQL Server 2014
  - SQL Server 2016
  - SQL Server 2017 on Windows and Linux
  - SQL Server 2019 on Windows and Linux
  - Azure SQL Database
- [Azure Synapse Analytics](#) is an analytics service for data warehouses and big data systems. This tool uses Spark technologies and has deep integration with Power BI, Azure Machine Learning, and other Azure services.

## Data integrators

- [Azure Data Factory](#) is a hybrid data integration service. You can use this fully managed, serverless solution to create, schedule, and orchestrate ETL and [ELT](#) workflows.
- [Azure Synapse Analytics](#) is an enterprise analytics service that accelerates time to insight, across data warehouses and big data systems. Azure Synapse brings together the best of SQL technologies (that are used in enterprise data warehousing), Spark technologies used for big data, Data Explorer for log and time series analytics, Pipelines for data integration and ETL/ELT, and deep integration with other Azure services, such as Power BI, Azure Cosmos DB, and Azure Machine Learning.
- [SQL Server Integration Services \(SSIS\)](#) is a platform for building enterprise-level data integration and transformation solutions. You can use SSIS to manage, replicate, cleanse, and mine data.
- [Azure Databricks](#) is a data analytics platform. Based on the Apache Spark open-source distributed processing system, Azure Databricks is optimized for the Azure cloud platform. In an analytics workflow, Azure Databricks reads data from multiple sources and uses Spark to provide insights.

## Data storage

- [Azure SQL Database](#) is part of the [Azure SQL](#) family and is built for the cloud. This service offers all the benefits of a fully managed and evergreen platform as a service. SQL Database also provides AI-powered, automated features that optimize performance and durability. Serverless compute and [Hyperscale storage options](#) automatically scale resources on demand.

- [SQL Managed Instance](#) is part of the Azure SQL service portfolio. This intelligent, scalable, cloud database service combines the broadest SQL Server engine compatibility with all the benefits of a fully managed and evergreen platform as a service. With SQL Managed Instance, you can modernize existing apps at scale.
- [SQL Server on Azure VMs](#) provides a way to lift and shift SQL Server workloads to the cloud with 100 percent code compatibility. As part of the Azure SQL family, SQL Server on Azure VMs offers the combined performance, security, and analytics of SQL Server with the flexibility and hybrid connectivity of Azure. With SQL Server on Azure VMs, you can migrate existing apps or build new apps. You can also access the latest SQL Server updates and releases, including SQL Server 2019.
- [Azure Database for PostgreSQL](#) is a fully managed relational database service that's based on the community edition of the open-source [PostgreSQL](#) database engine. With this service, you can focus on application innovation instead of database management. You can also scale your workload quickly and easily.
- [Azure Cosmos DB](#) is a globally distributed, [multimodel](#) database. With Azure Cosmos DB, your solutions can elastically and independently scale throughput and storage across any number of geographic regions. This fully managed [NoSQL](#) database service guarantees single-digit millisecond latencies at the ninety-ninth percentile anywhere in the world.
- [Data Lake Storage](#) is a storage repository that holds a large amount of data in its native, raw format. Data lake stores are optimized for scaling to terabytes and petabytes of data. The data typically comes from multiple, heterogeneous sources and may be structured, semi-structured, or unstructured. [Data Lake Storage Gen2](#) combines Data Lake Storage Gen1 capabilities with Blob Storage. This next-generation data lake solution provides file system semantics, file-level security, and scale. But it also offers the tiered storage, high availability, and disaster recovery capabilities of Blob Storage.
- [Azure Database for MariaDB](#) is a cloud-based relational database service. This service is based on the [MariaDB](#) community edition database engine.
- [Azure Database for MySQL](#) is a fully managed relational database service based on the [community edition of the open-source MySQL database engine](#).
- [Blob Storage](#) provides optimized cloud object storage that manages massive amounts of unstructured data.

## Networking

- An [on-premises data gateway](#) acts as a bridge that connects on-premises data with cloud services. Typically, you [install the gateway on a dedicated on-premises VM](#). Cloud services can then securely use on-premises data.
- An [IR](#) is the compute infrastructure that Data Factory uses to integrate data across different network environments. Data Factory uses [self-hosted IRs](#) to copy data between cloud data stores and data stores in on-premises networks. You can also use [Azure Synapse Pipelines](#).

## Scenario details

Data availability and integrity play an important role in mainframe and midrange modernization. [Data-first strategies](#) help to keep data intact and available during migration to Azure. To avoid impacting applications during modernization, sometimes you need to replicate data quickly or keep on-premises data in sync with Azure databases.

Specifically, this solution covers:

- Extraction: Connecting to and extracting from a source database.
- Transformation:
  - Staging: Temporarily storing data in its original format and preparing it for transformation.
  - Preparation: Transforming and manipulating data by using mapping rules that meet target database requirements.
- Loading: Inserting data into a target database.

## Potential use cases

Data replication and sync scenarios that can benefit from this solution include:

- Command Query Responsibility Segregation (CQRS) architectures that use Azure to service all inquire channels.
- Environments that test on-premises applications and rehosted or re-engineered applications in parallel.
- On-premises systems with tightly coupled applications that require phased remediation or modernization.

## Recommendations

When you use Data Factory to extract data, take steps to [tune the performance of the copy activity](#).

## Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Keep these points in mind when considering this architecture.

## Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

- Infrastructure management, including [availability](#), is automated in Azure databases.
- See [Pooling and failover](#) for information on the failover protection that Microsoft Service for DRDA provides.
- You can cluster the on-premises data gateway and IR to provide higher availability guarantees.

## Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

- Make use of [network security groups](#) to limit access of services to only what they need to function.
- Use [private endpoints](#) for your PaaS (Platform as a Service) services. Use service firewalls to supplement security for your services that are both reachable and unreachable through the Internet.
- Be aware of the differences between on-premises client identities and client identities in Azure. You will need to compensate for any differences.
- Use managed identities for component-to-component data flows.

- See [Planning and Architecting Solutions Using Microsoft Service for DRDA](#) to learn about the types of client connections that Microsoft Service for DRDA supports. Client connections affect the nature of transactions, pooling, failover, authentication, and encryption on your network.

## Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

- Pricing models vary between component services. Review the pricing models of the available component services to ensure the pricing models fit your budget.
- Use the [Azure pricing calculator](#) to estimate the cost of implementing this solution.

## Operational excellence

Operational excellence covers the operations processes that deploy an application and keep it running in production. For more information, see [Overview of the operational excellence pillar](#).

- Infrastructure management, including [scalability](#), is automated in Azure databases.
- You can [scale out the self-hosted IR](#) by associating the logical instance with multiple on-premises machines in active-active mode.
- You can cluster the on-premises data gateway and IR for scalability.

## Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

- When you use an on-premises application gateway, be aware of [limits on read and write operations](#).
- Consider [Azure ExpressRoute](#) as a high-scale option if your implementation uses significant bandwidth for initial replication or ongoing changed data replication.

- The [self-hosted IR](#) can only run on a Windows operating system.

## Next steps

- Contact [Azure Data Engineering - On-premises Modernization](#) for more information.
- Read the [Migration guide](#).

## Related resources

- [\[Azure data architecture guide\]](#)
- [Azure data platform end-to-end](#)

# Apache NiFi monitoring with MonitoFi

Azure Container Instances

Azure Container Registry

Azure Monitor

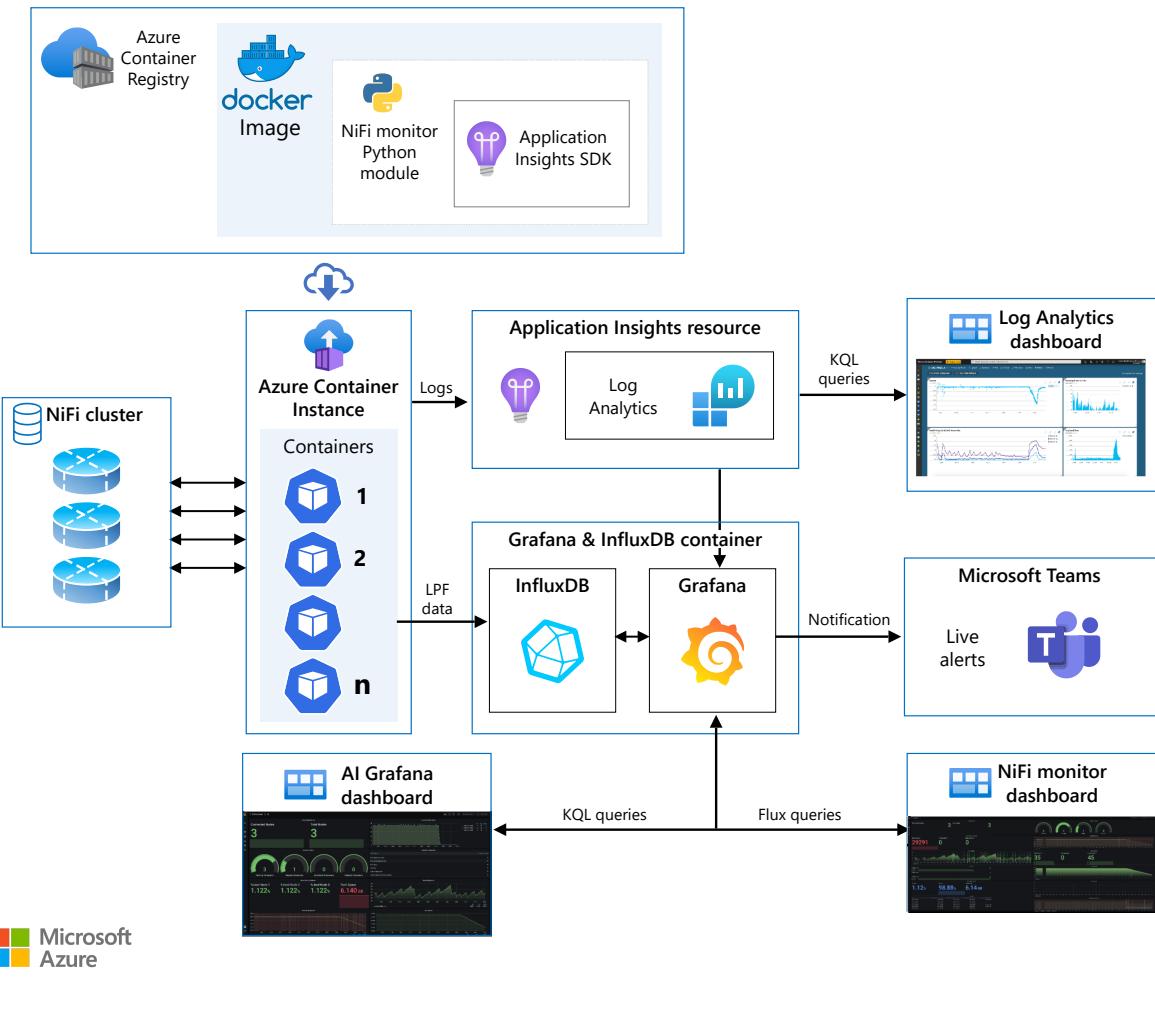
## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This solution monitors deployments of Apache NiFi on Azure by using MonitoFi. The tool sends alerts and displays health and performance information in dashboards.

*Apache®, Apache NiFi®, and NiFi® are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.*

## Architecture



Download a [Visio file](#) of this architecture.

## Workflow

- A Docker image encapsulates a MonitoFi Python module and the [Application Insights](#) SDK. This Docker image can be retrieved from the [Docker Hub registry](#) and stored in [Container Registry](#) for use and deployment.
- If [Container Instances](#) or [Docker](#) run on a local machine, the image can be retrieved to run instances of the MonitoFi container.
- Another container that hosts an InfluxDB server and a Grafana instance is deployed locally.
- The MonitoFi container collects information on each NiFi cluster's health and performance. The container requests data:
  - From clusters at configurable time intervals.
  - From various endpoints by using the [Apache NiFi REST API](#).
- The MonitoFi container converts the cluster data into these formats:
  - A structured log format. The container sends this data to Application Insights.

- InfluxDB line protocol. In air-gapped or on-premises environments, the container stores this data in a local instance of InfluxDB.
- Grafana displays Application Insights data. This data-monitoring tool:
  - Uses Monitor as a data source.
  - Runs [Kusto Query Language](#) queries. The Application Insights dashboard includes sample queries.
- Grafana is used to display data from the local instance of InfluxDB. For querying, Grafana uses the following languages:
  - The [Flux](#) query language
  - The [Influx Query Language \(InfluxQL\)](#)
- The Grafana notification system sends real-time alerts through email and [Microsoft Teams](#) when it detects anomalies in the cluster.

## Components

- MonitoFi runs in a [Docker](#) container, separately from NiFi.
- [Azure Container Registry](#) and [Azure Container Instances](#) manage and run the container images.

Other architecture components include:

- [Application Insights](#). This [Azure Monitor](#) feature monitors application usage, availability, and performance.
- [Grafana](#). This open-source analysis tool displays data and sends alerts.
- [InfluxDB](#). This platform stores data locally.

## Scenario details

[MonitoFi](#) is a tool that monitors the health and performance of [Apache NiFi](#) clusters. When you run [NiFi on Azure](#) and use MonitoFi:

- MonitoFi dashboards display historic information on the state of NiFi clusters.
- Real-time notifications alert users when anomalies are detected in clusters.

## Key benefits

MonitoFi has these advantages:

- Lightweight and extensible: MonitoFi is a lightweight tool that runs externally. Because MonitoFi is based on Python and is containerized, you can extend it to

add features. A MonitoFi instance that runs in one container can target multiple NiFi clusters.

- Effective and useful: MonitoFi uses local instances of InfluxDB and Grafana to provide real-time monitoring and alerts. MonitoFi can monitor clusters with latencies as low as one second.
- Flexible and robust: MonitoFi uses a REST API wrapper to retrieve JSON data from NiFi. MonitoFi converts that data into a usable format that doesn't depend on specific endpoints or field names. As a result, when NiFi REST API responses change, you don't need to change MonitoFi code.
- Easy to adopt: You don't have to reconfigure NiFi clusters to monitor them.
- Easy to use: MonitoFi offers preset configurations. It also includes templates for Grafana dashboards that you can import without modification.
- Highly configurable: MonitoFi runs in a Docker container. You configure MonitoFi by using environment variables. You can easily configure the following settings and others at runtime:
  - Endpoints
  - Settings for secure access
  - Certificates
  - Instrumentation key settings
  - The collection interval

With one container image, you can target different NiFi clusters, configurations, and different instances of Application Insights or InfluxDB. To change targets, you change the runtime command.

## Deploy this scenario

To deploy this solution, see [MonitoFi: Health & Performance Monitor for Apache NiFi on GitHub](#).

## Deployment examples

- In air-gapped and on-premises environments, there's no access to the public internet. As a result, these systems deploy a local instance of InfluxDB with Grafana. This approach provides a storage solution for the data. The MonitoFi container uses the NiFi REST API over a private IP address to retrieve cluster data. The

container stores this data in InfluxDB. Grafana is used to display the InfluxDB data and send email and Teams messages to alert users.

- In public environments, the MonitoFi container uses the NiFi REST API to retrieve cluster data. The container then sends this data in a structured format to Application Insights. These environments also deploy a local instance of InfluxDB and a Grafana container. MonitoFi can store data in that instance of InfluxDB. Grafana is used to display the data and send email and Teams messages to alert users.

## Deployment process

MonitoFi includes a fully automated deployment script that:

- Verifies prerequisites and installs missing dependencies.
- Deploys a MonitoFi Docker container.
- Deploys containers for InfluxDB and Grafana.
- Configures databases and a retention policy for InfluxDB.
- Configures a data source in Grafana for InfluxDB.
- Optionally configures a data source in Grafana for Monitor.
- Imports the MonitoFi dashboard into Grafana. Grafana uses this dashboard to access InfluxDB data.
- Optionally imports the Application Insights dashboard into Grafana. Grafana can use this dashboard to access Application Insights data.
- Configures a notification channel that Grafana uses for real-time Teams alerts.

## Deployment considerations

When you deploy this solution, keep in mind the following prerequisites and limitations:

- MonitoFi needs access to the NiFi cluster. Use one of these approaches to provide that access:
  - Place MonitoFi in the same network as the NiFi cluster. Provide access through a private IP address.
  - Make the NiFi cluster publicly accessible over the internet.
- The NiFi cluster can be secure or unsecured. For signing in, secured clusters support certificates in PKCS #12 format. Mount this type of certificate in the MonitoFi container, and make the password available.
- One MonitoFi instance can monitor multiple NiFi clusters at the same time. Another possibility is using multiple MonitoFi containers. In this case, the

containers can monitor different REST API endpoints in the same cluster or in different clusters.

- If you use more than one MonitoFi instance, it's possible to store the MonitoFi data in one InfluxDB database or send it to one common Application Insights resource. Pre-set tags mark the data and provide a way to identify its source.
- InfluxDB and Grafana run within the same Docker container. To provide a way for MonitoFi to send data to this container, use one of these options:
  - Place the Docker container in the same network as the MonitoFi container.
  - Make the Docker container publicly available.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Muazma Zahid](#) | Principal PM Manager

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [MonitoFi: Health & Performance Monitor for Apache NiFi on GitHub](#)
- [Docker Image with InfluxDB and Grafana](#)
- [MonitoFi: Health & Performance Monitor for Apache NiFi on Docker Hub](#)
- [Docker Image with InfluxDB and Grafana on Docker Hub](#)
- [NiFi Rest API 1.14.0](#)

## Related resources

- [Apache NiFi on Azure](#)
- [Helm-based deployments for Apache NiFi](#)
- [Monitoring Azure Functions and Event Hubs](#)
- [Web application monitoring on Azure](#)

# Augment security, observability, and analytics by using Microsoft Sentinel, Azure Monitor, and Azure Data Explorer

Azure Data Explorer

Azure Monitor

Microsoft Sentinel

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

Microsoft Sentinel, Azure Monitor, and Azure Data Explorer are based on a common technology and use Kusto Query Language (KQL) to analyze large volumes of data streamed in from multiple sources in near-real time.

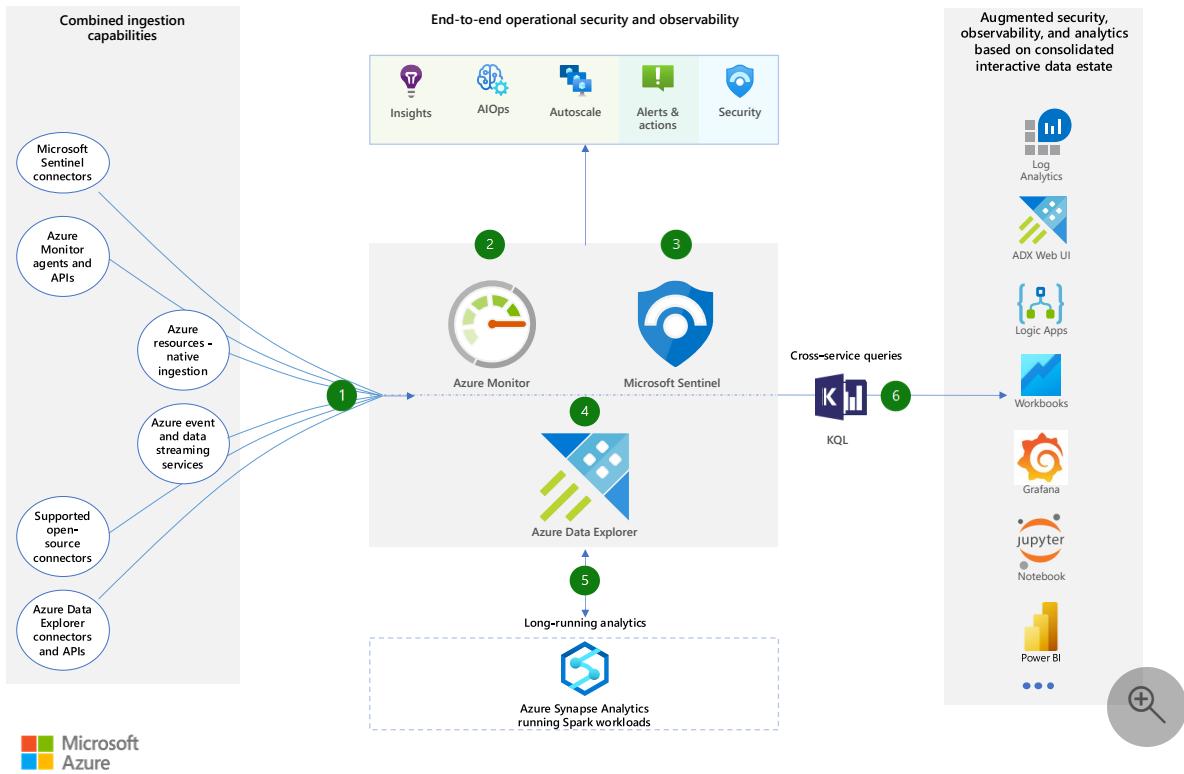
This solution demonstrates how to take advantage of the tight integration between Microsoft Sentinel, Azure Monitor, and Azure Data Explorer. You can use these services to consolidate a single interactive data estate and augment your monitoring and analytics capabilities.

## ⓘ Note

This solution applies to Azure Data Explorer and also to [Real-Time Analytics KQL databases](#), which provide SaaS-grade real-time log, time-series, and advanced analytics capabilities as part of [Microsoft Fabric](#).

*The Grafana and Jupyter logos and are trademarks of their respective companies. No endorsement is implied by the use of these marks.*

## Architecture



Download a [PowerPoint file](#) of this architecture.

## Dataflow

1. Ingest data by using the combined ingestion capabilities of [Microsoft Sentinel](#), [Azure Monitor](#), and [Azure Data Explorer](#):
  - Configure diagnostic settings to ingest data from Azure services like Azure Kubernetes Service (AKS), Azure App Service, Azure SQL Database, and Azure Storage.
  - Use Azure Monitor Agent to ingest data from VMs, containers, and workloads.
  - Use a wide range of connectors, agents, and APIs supported by the three services to ingest data from on-premises resources and other clouds. Supported connectors, agents, and APIs include Logstash, Kafka, and Logstash connectors, OpenTelemetry agents, Azure Data Explorer APIs, and the Azure Monitor Log Ingestion API.
  - Stream data in by using Azure services like Azure IoT Hub, Azure Event Hubs, and Azure Stream Analytics.
2. Use Microsoft Sentinel to monitor, investigate, and alert and act on security-related data across your IT environment.
3. Use Azure Monitor to monitor, analyze, and alert and act on the performance, availability, and health of applications, services, and IT resources. Doing so enables

you to gain insights into the operational status of your cloud infrastructure, identify problems, and optimize performance.

4. Use Azure Data Explorer for any data that requires custom or more flexible handling or analytics, including full schema control, cache or retention control, deep data platform integrations, and machine learning.
5. Optionally, apply advanced machine learning on a broad set of data from your entire data estate to discover patterns, detect anomalies, get forecasts, and gain other insights.
6. Take advantage of the tight integration between services to augment monitoring and analytics capabilities:
  - Run cross-service queries from [Microsoft Sentinel](#), [Monitor](#), and [Azure Data Explorer](#) to analyze and correlate data in all three services in one query without moving the data.
  - Consolidate a single-pane-of-glass view of your data estate with customized cross-service workbooks, dashboards, and reports.

## Components

Use cross-service queries to build a consolidated, interactive data estate, joining data in Microsoft Sentinel, Monitor, and Azure Data Explorer:

- [Microsoft Sentinel](#) is the Azure cloud-native solution for security information and event management (SIEM) and security orchestration, automation, and response (SOAR). Microsoft Sentinel has the following features:
  - Connectors and APIs for collecting security data from various sources, like Azure resources, Microsoft 365, and other cloud and on-premises solutions.
  - Advanced built-in analytics, machine learning, and threat intelligence capabilities for detecting and investigating threats.
  - Rules-based case management and incident response automation capabilities that use modular, reusable playbooks that are based on Azure Logic Apps.
  - KQL query capabilities that let you analyze security data and hunt for threats by correlating data from multiple sources and services.
- [Azure Monitor](#) is the Azure managed solution for IT and application monitoring. Monitor has the following features:
  - Native ingestion of monitoring data from Azure resources. Agents, connectors, and APIs for collecting monitoring data from Azure resources and any sources, applications, and workloads in Azure and hybrid environments.

- IT monitoring tools and analytics features, including AI for IT operations (AIOps) features, alerting and automated actions, and prebuilt workbooks for monitoring specific resources, like virtual machines, containers, and applications.
- End-to-end observability capabilities that help you improve IT and application efficiency and performance.
- KQL query capabilities that enable you to analyze data and troubleshoot operational issues by correlating data across resources and services.
- [Azure Data Explorer](#) is part of the Azure data platform. It provides real-time advanced analytics for any type of structured and unstructured data. It has the following features:
  - Connectors and APIs for various types of IT and non-IT data, for example, business, user, and geospatial data.
  - The full set of KQL's analytics capabilities, including hosting of machine learning algorithms in Python and federated queries to other data technologies, like SQL Server, data lakes, and Azure Cosmos DB.
  - Scalable data management capabilities, including full schema control, processing of incoming data by using KQL, materialized views, partitioning, granular retention, and caching controls.
  - Cross-service query capabilities that enable you to correlate collected data with data in Microsoft Sentinel, Monitor, and other services.

## Scenario details

An architecture built on the features and flexibility provided by Microsoft Sentinel, Monitor, and Azure Data Explorer gives you:

- A broad range of data ingestion options that span various types of data and data sources.
- A powerful set of native security, observability, and data analytics features and capabilities.
- The ability to use cross-service queries to create a single-pane-of-glass view of your data by:
  - Querying IT monitoring and non-IT data.
  - Applying machine learning on a broad dataset to discover patterns, implement anomaly detection and forecasting, and get other advanced insights.
  - Creating workbooks and reports that enable you to monitor, correlate, and act on various types of data.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Guy Wild](#) | Senior Content Developer

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [Azure Data Explorer documentation](#)
- [Training: Introduction to Azure Data Explorer](#)
- [Azure Monitor overview](#)
- [What is Microsoft Sentinel?](#)

## Related resources

- [Big data analytics with Azure Data Explorer](#)
- [Azure Data Explorer interactive analytics](#)
- [IoT analytics with Azure Data Explorer](#)

# Data management across Azure Data Lake with Microsoft Purview

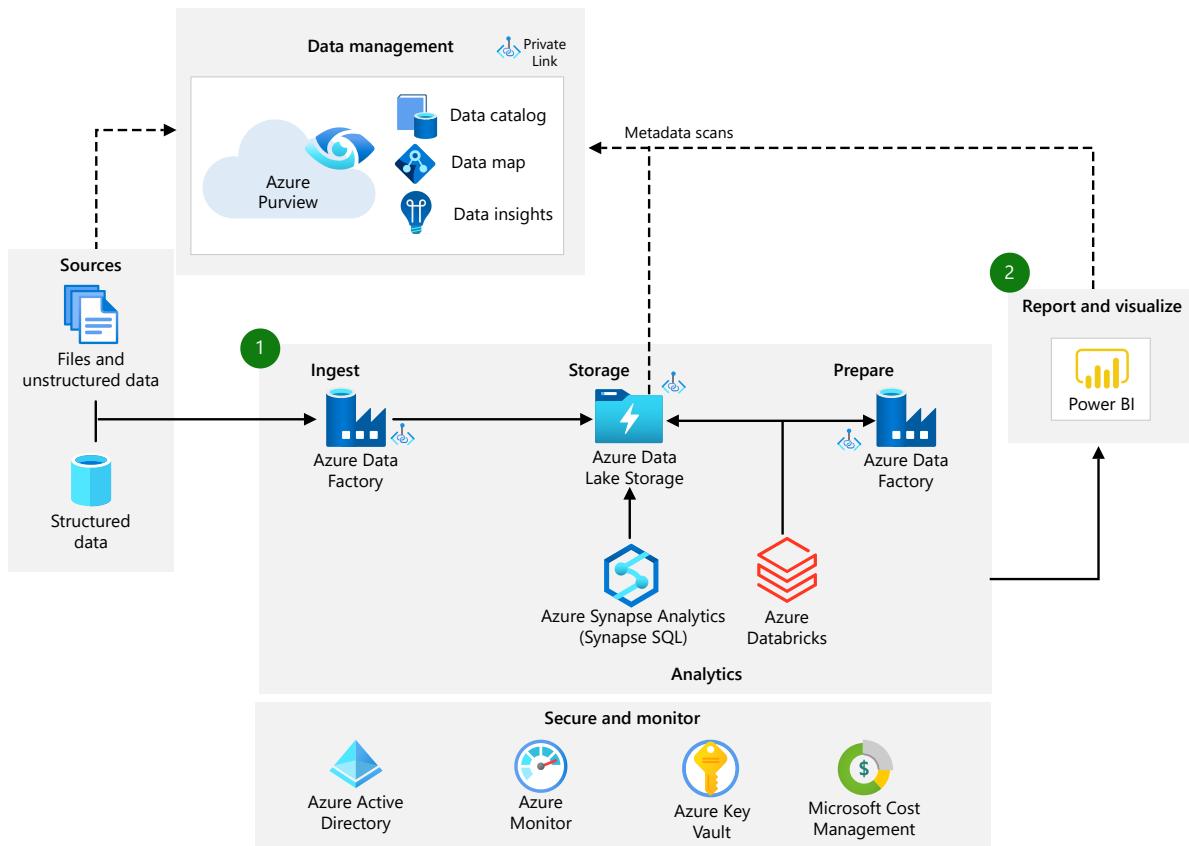
Azure Data Factory Microsoft Purview Azure Data Lake Storage Azure Synapse Analytics Power BI

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes a solution that uses Azure Purview to build a foundation for data governance and management that can produce and deliver high-quality, trusted data.

## Architecture



 Microsoft Azure

Download a [Visio file](#) of this architecture.

# Dataflow

Azure Purview provides a single, unified data management service for the data from all sources, in the data lake, and in end reporting tools.

Scenarios for connecting Azure Purview to Data Lake services:

1. Azure Purview provides an improved-security connection to your data lake ingestion, storage, and analytics pipelines to automatically catalog data assets. It also provides lineage across these services. Specific Azure services include Data Factory, Data Lake Storage, and Azure Synapse Analytics.
2. Azure Purview connects natively with Power BI and other reporting and visualization tools. It shows the lineage of data that's used in end reports. It also shares sensitivity information from the Power BI assets to prevent incorrect data use.

## Important

The information that's transferred from the sources to Azure Purview is metadata that describes the data within the scanned sources. No actual data is transferred from the sources to Azure Purview.

## Capabilities

- **Catalog.** The Azure Purview Data Catalog can automatically capture and describe core characteristics of data at the source, including schema, technical properties, and location. The Azure Purview glossary allows a business-friendly definition of data to be layered on top, to improve search and discovery.
- **Classification.** Azure Purview automatically classifies datasets and data elements with 100 predefined sensitive-data classifications. It also allows you to define your own custom classification schemes that you can apply manually and automatically.
- **Lineage.** Azure Purview diagrammatically visualizes lineage across Data Factory, Azure Synapse Analytics, and Power BI pipelines. These visualizations show the end-to-end flow of data at a granular level.
- **Access control.** Azure Purview access control policy allows you to define and grant access to data assets from the catalog, directly on the underlying sources.
- **Ownership.** Azure Purview allows you to apply data ownership and stewardship to data assets and glossary items in the catalog.

- [Insight](#). Insights in Azure Purview provide multiple predefined reports to help CDOs, data professionals, and data governance professionals gain a detailed understanding of the data landscape.

## Components

- [Azure Purview](#) is a unified data catalog that manages on-premises, multicloud, and software as a service (SaaS) data. This data governance service maintains data landscape maps. Features include automated data discovery, sensitive data classification, and data lineage.
- [Data Factory](#) is a fully managed, serverless data integration service that helps you construct ETL and ELT processes.
- [Data Lake Storage](#) provides massively scalable, high-security, cost-effective cloud storage for high-performance analytics workloads.
- [Azure Synapse Analytics](#) is a limitless analytics service that brings together data integration, enterprise data warehousing, and big data analytics.
- [Power BI](#) is a collection of software services and apps. These services create and share reports that connect and visualize multiple sources of data. When you use Power BI with Azure Purview, it can catalog and classify your data and provide granular lineage that's illustrated from end to end.
- [Azure Private Link](#) provides private connectivity from a virtual network to Azure platform as a service (PaaS) services, services that you own, or Microsoft partner services.
- [Azure Key Vault](#) stores and controls access to secrets like tokens, passwords, and API keys. Key Vault also creates and controls encryption keys and manages security certificates.
- [Microsoft Entra ID](#) offers cloud-based identity and access management services. These features provide a way for users to sign in and access resources.
- [Azure Monitor](#) collects and analyzes data on environments and Azure resources. This data includes app telemetry, like performance metrics and activity logs.

## Scenario details

As you load more data into Azure, the need to properly govern and manage that data across all your data sources and data consumers also grows.

If you don't have high-quality data in your Azure data estate, the business value of Azure is diminished. The solution is to build a foundation for data governance and management that can produce and deliver high-quality, trusted data.

Data needs to be managed at scale across on-premises, cloud, and multicloud storage to ensure it meets compliance requirements for security, privacy, and usage. Well-managed data can also improve self-discovery, data sharing, and data quality, which improves the use of data in applications and analytics.

[Azure Purview](#) provides governance for finding, classifying, defining, and enforcing policies and standards across data. You can use it to apply definitions, classifications, and governance processes uniformly across data. It catalogs all data sources, identifies any sensitive information, and defines data lineage. It provides a central platform where you can apply definitions and ownership to data. With a single view on reports and insight, it can help you generate data standards that should be applied to your data.

Working with other Azure services, Azure Purview can automatically discover, catalog, classify, and manage data across Azure Data Lake offerings and partner services.

## Potential use cases

The requirements for data management differ across industries. For all industries, the need to govern data at scale has increased as the size and complexity of data and data architectures grow. This is appropriate for organizations that would benefit from the following outcomes of well-governed data:

- Automatic discovery of data to accelerate cloud adoption.
- Improved security of data for compliance with data laws and regulations.
- Improved access, discovery, and quality of managed data to enhance analytics.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Isabel Arevalo](#) ↗ | Senior Cloud Solution Architect

## Next steps

- [Azure Purview customer case studies](#) ↗

- Microsoft Purview technical documentation and best practices
- What is Microsoft Purview?
- What is Power BI? ↗
- What is Microsoft Entra ID?
- What is Data Factory?
- Introduction to Data Lake Storage
- What is Azure Databricks?
- Azure Monitor overview
- What is Azure Synapse Analytics?
- What is Azure Key Vault?

## Related resources

- Data analysis workloads for regulated industries
- Query a data lake or lakehouse by using Azure Synapse serverless
- Choose an analytical data store in Azure
- Choose a data analytics technology in Azure

# TimeXtender with cloud scale analytics

Azure Analysis Services

Azure Data Lake Storage

Azure Databricks

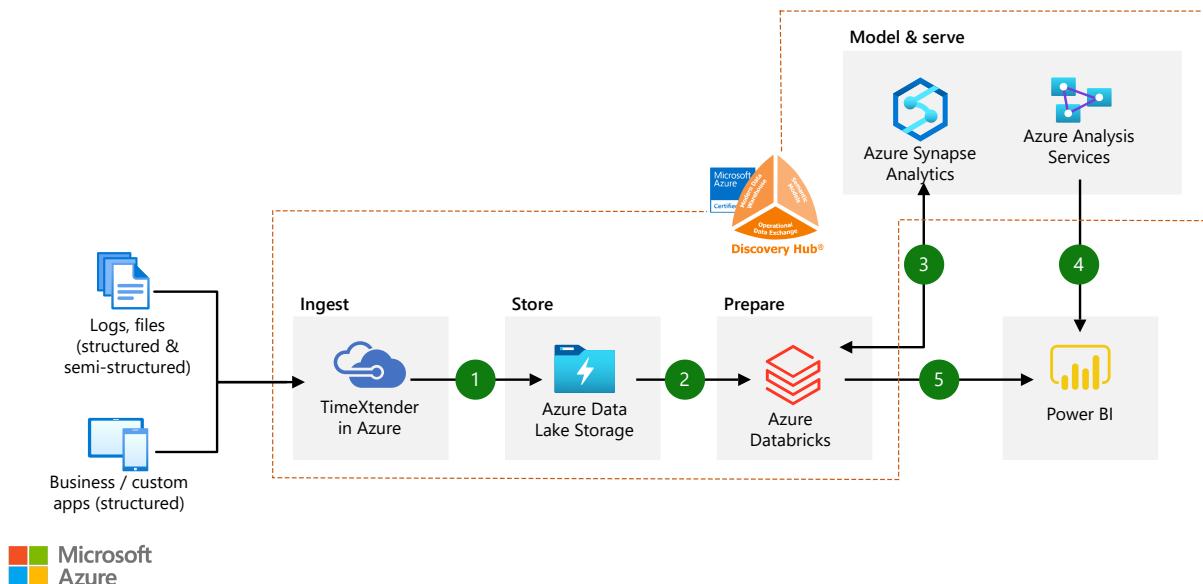
Azure Synapse Analytics

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This solution idea describes how to use the TimeXtender graphical interface to define a data estate.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

1. Combine all your structured and semi-structured data in Azure Data Lake Storage using TimeXtender's data engineering pipeline with hundreds of native data connectors.

2. Clean and transform data using the powerful analytics and computational ability of Azure Databricks.
3. Move cleansed and transformed data to Azure Synapse Analytics, creating one hub for all your data. Take advantage of native connectors between Azure Databricks (PolyBase) and Azure Synapse Analytics to access and move data at scale.
4. Build operational reports and analytical dashboards on top of SQL Database to derive insights from the data and use Azure Analysis Services to serve the data.
5. Run ad-hoc queries directly on data within Azure Databricks.

## Components

- [Azure Data Lake Storage](#): Massively scalable, secure data lake functionality built on Azure Blob Storage
- [Azure Databricks](#): Fast, easy, and collaborative Apache Spark-based analytics platform
- [Azure Synapse Analytics](#): Limitless analytics service with unmatched time to insight (formerly SQL Data Warehouse)
- [Azure Analysis Services](#): Enterprise-grade analytics engine as a service
- [Power BI Embedded](#): Embed fully interactive, stunning data visualizations in your applications

## Scenario details

You can use TimeXtender to define a data estate via a graphical user interface. Definitions are stored in a metadata repository. Code for building the data estate is generated automatically while remaining fully customizable. The results are a modern data warehouse that is ready to support cloud scale analytics and AI.

## Potential use cases

- No infrastructure issues or maintenance
- Consistent performance
- Deploy and manage both the architecture and the data pipelines, data models and semantic models

## Next steps

- [Azure Data Lake Storage documentation](#)
- [Azure Databricks documentation](#)
- [Azure Synapse Analytics documentation](#)

- [Azure Analysis Services documentation](#)
- [Power BI Embedded documentation](#)

## Related resources

- [Modern data warehouse for small and medium business](#)
- [Data warehousing and analytics](#)
- [Modern analytics architecture with Azure Databricks](#)

# Ingestion, ETL, and stream processing pipelines with Azure Databricks and Delta Lake

Azure Databricks

Azure Data Lake Storage

Azure IoT Hub

Azure Data Factory

Azure Event Hubs

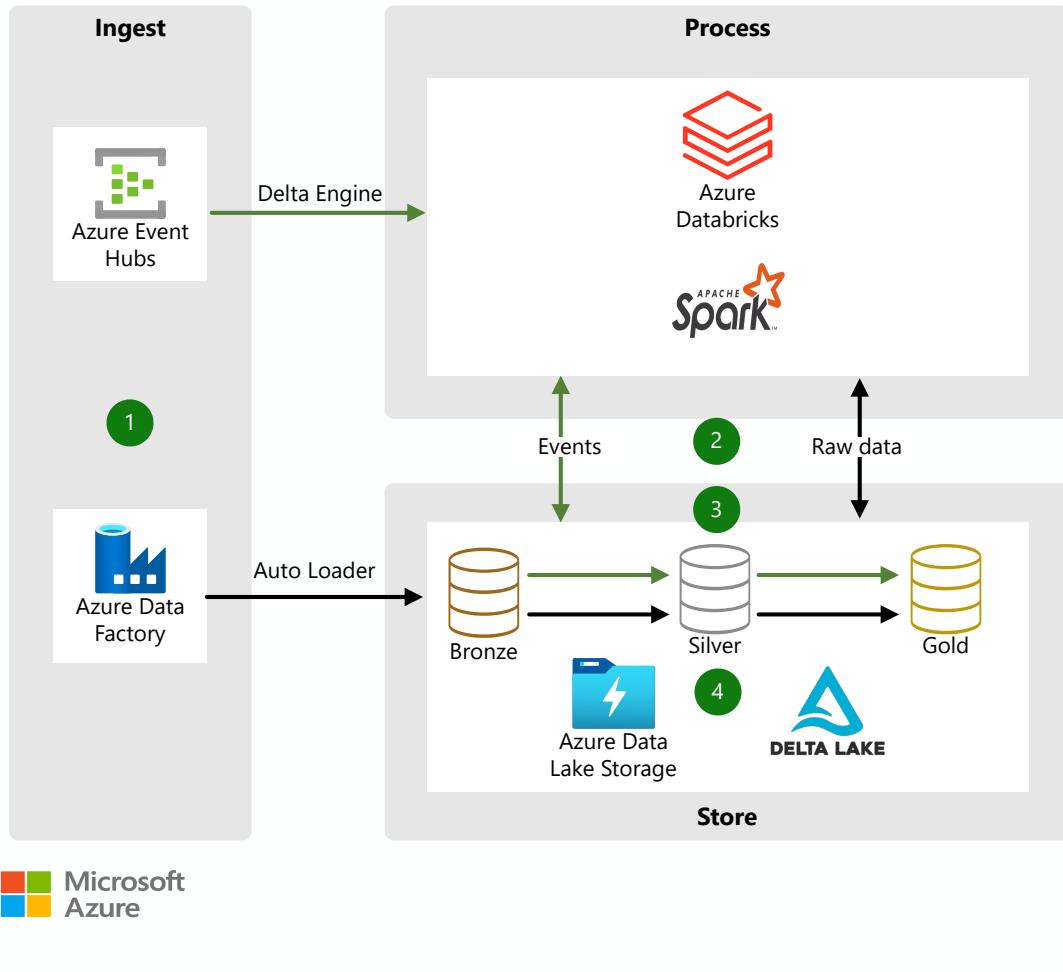
## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

Your organization needs to ingest data of any format, size, and speed into the cloud in a consistent way. The solution in this article meets that need with an architecture that implements extract, transform, and load (ETL) from your data sources to a data lake. The data lake can hold all the data, including transformed and curated versions at various scales. The data can be used for data analytics, business intelligence (BI), reporting, data science, and machine learning.

*Apache® and Apache Spark™ are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.*

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

1. Data is ingested in the following ways:
  - Event queues like Event Hubs, IoT Hub, or Kafka send streaming data to Azure Databricks, which uses the optimized Delta Engine to read the data.
  - Scheduled or triggered Data Factory pipelines copy data from different data sources in raw formats. The [Auto Loader in Azure Databricks](#) processes the data as it arrives.
2. Azure Databricks loads the data into optimized, compressed Delta Lake tables or folders in the Bronze layer in Data Lake Storage.
3. Streaming, scheduled, or triggered Azure Databricks jobs read new transactions from the Data Lake Storage Bronze layer. The jobs join, clean, transform, and aggregate the data before using ACID transactions to load it into curated data sets in the Data Lake Storage Silver and Gold layers.

#### 4. The data sets are stored in Delta Lake in Data Lake Storage.

Each service ingests data into a common format to ensure consistency. The architecture uses a shared data lake based on the open Delta Lake format. Raw data is ingested from different batch and streaming sources to form a unified data platform. The platform can be used for downstream use cases such as analytics, BI reporting, data science, AI, and machine learning.

## Bronze, Silver, and Gold storage layers

With the medallion pattern, consisting of Bronze, Silver, and Gold storage layers, customers have flexible access and extendable data processing.

- **Bronze** tables provide the entry point for raw data when it lands in Data Lake Storage. The data is taken in its raw source format and converted to the open, transactional Delta Lake format for processing. The solution ingests the data into the Bronze layer by using:
  - Apache Spark APIs in Azure Databricks. The APIs read streaming events from Event Hubs or IoT Hub, and then convert those events or raw files to the Delta Lake format.
  - The **COPY INTO** command. Use the command to copy data directly from a source file or directory into Delta Lake.
  - The Azure Databricks Auto Loader. The Auto Loader grabs files when they arrive in the data lake and writes them to the Delta Lake format.
  - The Data Factory Copy Activity. Customers can use this option to convert the data from any of its supported formats into the Delta Lake format.
- **Silver** tables store data while it's being optimized for BI and data science use cases. The Bronze layer ingests raw data, and then more ETL and stream processing tasks are done to filter, clean, transform, join, and aggregate the data into Silver curated datasets. Companies can use a consistent compute engine, like the open-standards [Delta Engine](#), when using Azure Databricks as the initial service for these tasks. They can then use familiar programming languages like SQL, Python, R, or Scala. Companies can also use repeatable DevOps processes and ephemeral compute clusters sized to their individual workloads.
- **Gold** tables contain enriched data, ready for analytics and reporting. Analysts can use their method of choice, such as PySpark, Koalas, SQL, Power BI, and Excel to gain new insights and formulate queries.

## Components

- [Event Hubs](#) parses and scores streaming messages from various sources, including on-premises systems, and provides real-time information.
- [Data Factory](#) orchestrates data pipelines for ingestion, preparation, and transformation of all your data at any scale.
- [Data Lake Storage](#) brings together streaming and batch data, including structured, unstructured, and semi-structured data like logs, files, and media.
- [Azure Databricks](#) cleans and transforms the structureless data sets and combines them with structured data from operational databases or data warehouses.
- [IoT Hub](#) gives you highly secure and reliable communication between your IoT application and devices.
- [Delta Lake](#) on Data Lake Storage supports ACID transactions for reliability and is optimized for efficient ingestion, processing, and queries.

## Scenario details

Ingestion, ETL, and stream processing with Azure Databricks is simple, open, and collaborative:

- **Simple:** An open data lake with a curated layer in an open-source format simplifies the data architecture. Delta Lake, an open-source tool, provides access to the Azure Data Lake Storage data lake. Delta Lake on Data Lake Storage supports atomicity, consistency, isolation, and durability (ACID) transactions for reliability. Delta Lake is optimized for efficient ingestion, processing, and queries.
- **Open:** The solution supports open-source code, open standards, and open frameworks. It also works with popular integrated development environments (IDEs), libraries, and programming languages. Through native connectors and APIs, the solution works with a broad range of other services, too.
- **Collaborative:** Data engineers, data scientists, and analysts work together with this solution. They can use collaborative notebooks, IDEs, dashboards, and other tools to access and analyze common underlying data.

Azure Databricks seamlessly integrates with other Azure services like Data Lake Storage, Azure Data Factory, Azure Event Hubs, and Azure IoT Hub.

## Potential use cases

This solution is inspired by the system that [Providence Health Care](#) built for real-time analytics. Any industry that ingests batch or streaming data could also consider this solution. Examples include:

- Retail and e-commerce

- Finance
- Healthcare and life sciences
- Energy suppliers

## Next steps

- [Providence Health Care](#) builds their data streaming solution using Azure Databricks and Azure Event Hubs to improve the National Emergency Department Overcrowding Score for each of its emergency departments.
- [Spanish Point Technologies](#) builds its Matching Engine using Azure Databricks and Azure Data Factory to ingest data at scale to help musicians get paid fairly.

## Related resources

Guides and fully deployable architectures:

- [Choose an analytical data store in Azure](#)
- [Stream processing with Azure Databricks](#)
- [Automated enterprise BI](#)

# Mining equipment monitoring

Azure Analysis Services

Azure Logic Apps

Azure Data Lake Storage

Azure Databricks

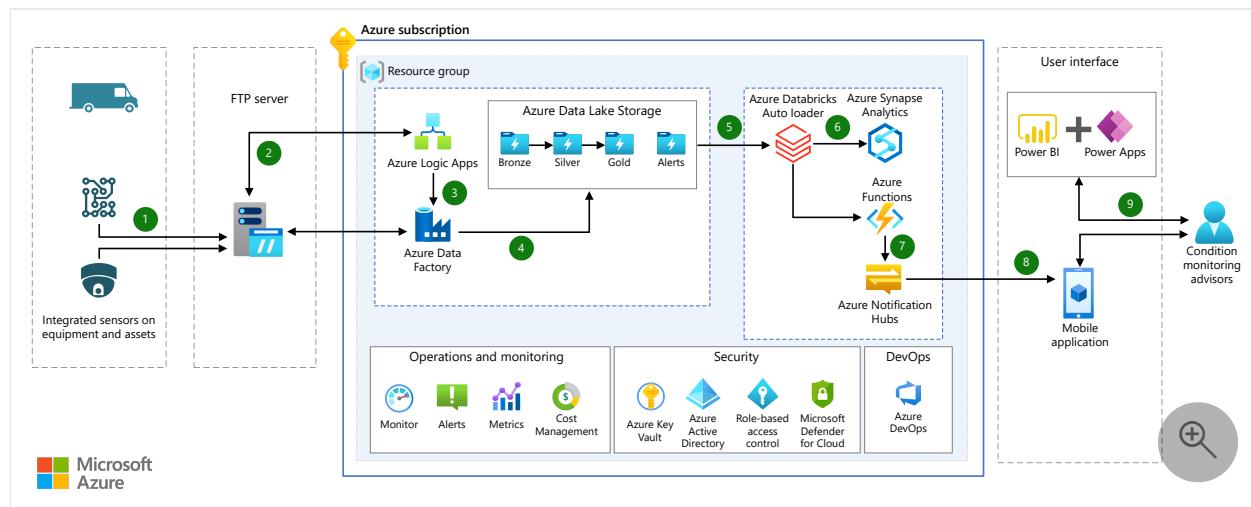
Azure Monitor

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

Mining companies can have Azure continually monitor the performance data from their equipment or from other assets. Analysis of the data identifies anomalies and results in recommendations for maintenance and repair. Such monitoring can prevent failures and reduce operating costs.

## Architecture



[Download a Visio file](#) of this architecture.

## Dataflow

The data flows through the solution as follows:

1. Equipment and other assets have integrated sensor systems that deliver sensor data (in CSV files) to a folder in an FTP server or to Azure Storage.
2. Azure Logic App monitors the folder for new or modified files.
3. Logic App triggers the Data Factory pipeline when a file is added to the folder, or when a previously added file is modified.

4. Azure Data Factory obtains the data from the FTP server or from Azure Storage, and stores it to a data lake that Azure Data Lake provides. The Delta Lake open-source software augments Data Lake capabilities.
5. The cloudFiles feature of Azure Databricks Auto Loader automatically processes new files as they arrive at the data lake, and can also process existing files.
  - a. cloudFiles uses structured streaming APIs to check if sensor values exceed thresholds. If so, it copies the values to a separate storage folder (Alerts).
  - b. After appropriate cleansing and transforming of the data, it moves the data to Delta Lake Bronze/Silver/Gold folders. The folders contain various transformations of the data; for example, ingested (Bronze), to refined (Silver), to aggregated (Gold).
6. An Azure Synapse connector in Azure Databricks moves the data from the data lake to an Azure Synapse Analytics dedicated SQL pool.
7. Whenever a new alert arrives in the Alerts folder, Azure Function Apps sends notifications to Azure Notification Hub.
8. Notification Hub then sends notifications to various mobile platforms to alert operators and administrators of events that require attention.
9. Monitoring advisors can create visual reports to explore the data. They can publish and share them, and collaborate with others. Power BI integrates with other tools, including Power Apps. Advisors can integrate Power BI reports into a Canvas App in Power Apps for a good user experience.

## Components

Data is loaded from these different data sources using several Azure components:

- [Azure Data Lake Storage](#) makes Azure Storage the foundation for building enterprise data lakes on Azure. It can quickly process massive amounts of data (petabytes).
- [Azure Data Factory](#) is a managed service that orchestrates and automates data movement and data transformation. In this architecture, it copies the data from the source to Azure Storage.
- [Azure Logic Apps](#) are automated workflows for common enterprise orchestration tasks. Logic Apps includes [connectors](#) for many popular cloud services, on-premises products, and other applications.
- [Azure Databricks](#) is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Databricks is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive workspace that was designed in collaboration with the founders of Apache Spark.
- [Azure Databricks – Auto Loader](#) provides a structured streaming source called cloudFiles. The cloudFiles source automatically processes new files as they arrive at

a directory, and can also process other files in the directory.

- [Azure Synapse Analytics](#) is a distributed system for storing and analyzing large datasets. Its use of massive parallel processing (MPP) makes it suitable for running high-performance analytics.
- [Azure Functions](#) allows you to run small pieces of code (called "functions") without worrying about application infrastructure. Azure Functions is a great solution for processing bulk data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and micro-services.
- [Power BI](#) is a suite of business analytics tools to analyze data and provide insights. Power BI can query a semantic model stored in Analysis Services, or it can query Azure Synapse directly.
- [Power Apps](#) is a suite of apps, services, and connectors for building custom business apps. It includes an underlying data platform ([Microsoft Dataverse](#)) and a rapid development environment.

## Scenario details

### Potential use cases

- Monitor mining equipment and other equipment that can provide the needed data. This solution is ideal for the energy industry.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Ansley Yeo](#) | Technology Leader and IoT

## Next steps

- [Create, monitor, and manage FTP files by using Azure Logic Apps](#)
- [Copy data from FTP server by using Azure Data Factory](#)
- [Load files from Azure Blob storage and Azure Data Lake Storage Gen1 and Gen2 using Auto Loader](#)
- [Azure Synapse Analytics](#)
- [On GitHub: azure-notificationhubs-dotnet/Samples/AzFunctions/](#)
- [Azure SQL Data Warehouse with DirectQuery](#)

- [Power Apps visual for Power BI](#)

Information about the Delta Lake open-source project for building a Lakehouse architecture:

- [Delta Lake Key Features ↗](#)
- [What is Delta Lake](#)
- [Delta Lake and Delta Engine guide](#)

## Related resources

See the following related database architectural guidance:

- [Non-relational data and NoSQL](#)
- [Big data architectures](#)
- [Choosing a batch processing technology in Azure](#)
- [Data lakes](#)
- [Choosing a big data storage technology in Azure](#)
- [Modernize mainframe & midrange data](#)
- [Master data management with Profisee and Azure Data Factory](#)
- [Master Data Management powered by CluedIn](#)
- [DataOps for the modern data warehouse](#)
- [Data warehousing and analytics](#)
- [Real Time Analytics on Big Data Architecture](#)

See the following related IoT architectural guidance:

- [IoT solutions conceptual overview](#)
- [Vision with Azure IoT Edge](#)
- [Azure Industrial IoT Analytics Guidance](#)
- [Azure IoT reference architecture](#)
- [IoT and data analytics](#)

# Tier applications and data for analytics

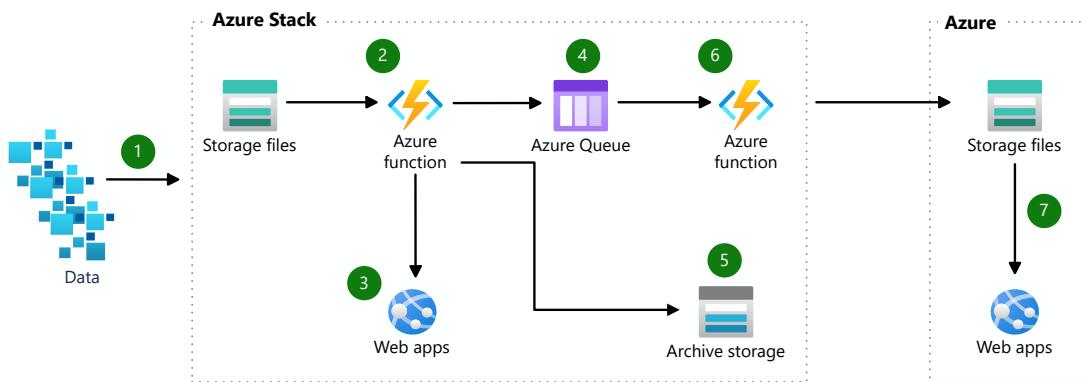
Azure Functions   Azure Stack   Azure Storage   Azure App Service

## 💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This solution idea describes how to tier data and applications on-premises and on Azure. As data flows into a storage account, you can use Azure Stack to analyze the data for anomalies or compliance and to display insights in apps.

## Architecture



Download a [Visio file](#) of this architecture.

## Dataflow

1. Data flows into a storage account.
2. Function on Azure Stack analyzes the data for anomalies or compliance.
3. Locally relevant insights are displayed on the Azure Stack app.
4. Insights and anomalies are placed into a queue.

5. The bulk of the data is placed into an archive storage account.
6. Function sends data from queue to Azure Storage.
7. Globally relevant and compliant insights are available in the global app.

## Components

- [Storage](#): Durable, highly available, and massively scalable cloud storage
- [Azure Functions](#): Process events with serverless code
- [Azure Stack](#): Build and run innovative hybrid applications across cloud boundaries

## Scenario details

This scenario can help you tier data and applications on-premises and on Azure. Filter unnecessary data early in the process, bring cloud applications close to the data on-premises, and analyze large-scale aggregate data from multiple locations on Azure.

## Potential use cases

Tiered applications provide the following benefits:

- The ability to update the technology stack of one tier without affecting other areas of the application.
- Development teams work on their own areas of expertise.
- Able to scale the application.
- Adds reliability and more independence of the underlying servers or services.

## Next steps

- [Storage documentation](#)
- [Azure Functions documentation](#)
- [Azure Stack documentation](#)

## Related resources

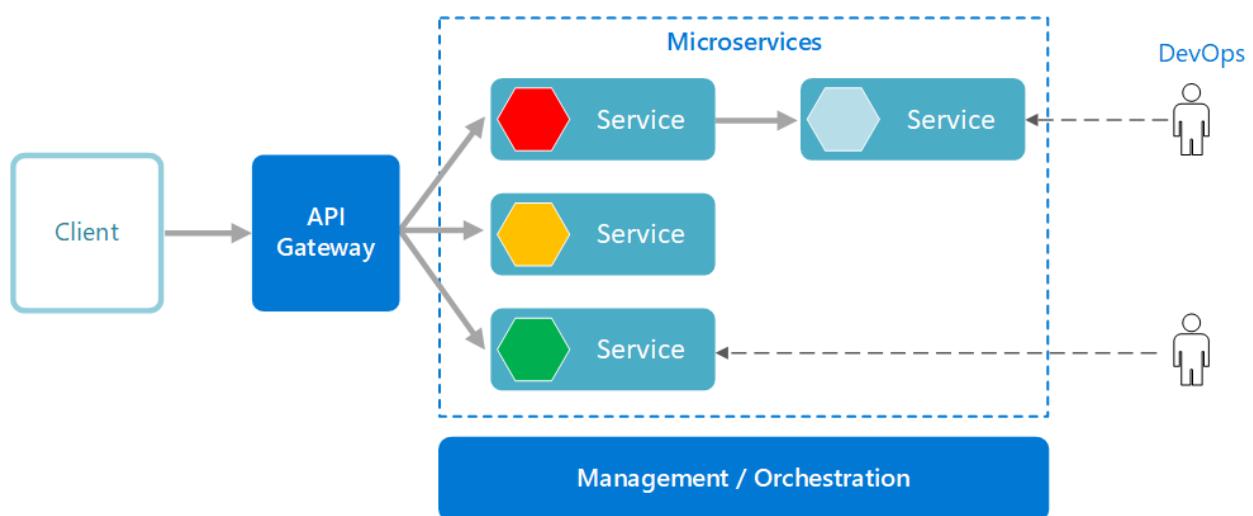
- [Analytics architecture design](#)
- [Data analysis workloads for regulated industries](#)
- [Tiered data for analytics](#)

# Microservices architecture design

Azure DevOps

Microservices are a popular architectural style for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. But a successful microservices architecture requires a different approach to designing and building applications.

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.



## What are microservices?

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.

- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

Besides for the services themselves, some other components appear in a typical microservices architecture:

**Management/orchestration.** This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

**API Gateway.** The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

## Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features might be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend to be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base.** In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching

code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.

- **Mix of technologies.** Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Fault isolation.** If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly. For example, you can implement the [Circuit Breaker pattern](#), or you can design your solution so that the microservices communicate with each other using [asynchronous messaging patterns](#).
- **Scalability.** Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation.** It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application might all touch the same data, making any alterations to the schema risky.

## Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- **Complexity.** A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing.** Writing a small service that relies on other dependent services requires a different approach than writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance.** The decentralized approach to building microservices has advantages, but it can also lead to problems. You might end up with so many different languages and frameworks that the application becomes hard to maintain. It might be useful to put some project-wide standards in place, without

overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

- **Network congestion and latency.** The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like [queue-based load leveling](#).
- **Data integrity.** With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- **Management.** To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- **Versioning.** Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- **Skill set.** Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

## Process for building a microservices architecture

The articles listed here present a structured approach for designing, building, and operating a microservices architecture.

**Domain analysis.** To avoid some common pitfalls when designing microservices, use domain analysis to define your microservice boundaries. Follow these steps:

1. [Use domain analysis to model microservices](#).
2. [Use tactical DDD to design microservices](#).
3. [Identify microservice boundaries](#).

**Design the services.** Microservices require a different approach to designing and building applications. For more information, see [Designing a microservices architecture](#).

**Operate in production.** Because microservices architectures are distributed, you must have robust operations for deployment and monitoring.

- CI/CD for microservices architectures
- Build a CI/CD pipeline for microservices on Kubernetes
- Monitor microservices running on Azure Kubernetes Service (AKS)

## Microservices reference architectures for Azure

- Microservices architecture on Azure Kubernetes Service (AKS)
- Microservices architecture on Azure Service Fabric

# Microservices assessment and readiness

Article • 09/19/2022

A microservices architecture can provide many benefits for your applications, including agility, scalability, and high availability. Along with these benefits, this architecture presents challenges. When you build microservices-based applications or transform existing applications into a microservices architecture, you need to analyze and assess your current situation to identify areas that need improvement.

This guide will help you understand some considerations to keep in mind when you move to a microservices architecture. You can use this guide to assess the maturity of your application, infrastructure, DevOps, development model, and more.

## Understand business priorities

To start evaluating a microservices architecture, you need to first understand the core priorities of your business. Core priorities might be related to agility, change adoption, or rapid development, for example. You need to analyze whether your architecture is a good fit for your core priorities. Keep in mind that business priorities can change over time. For example, innovation is a top priority for startups, but after a few years, the core priorities might be reliability and efficiency.

Here are some priorities to consider:

- Innovation
- Reliability
- Efficiency

Document the SLAs that are aligned with various parts of your application to ensure an organizational commitment that can serve as a guide to your assessment.

## Record architectural decisions

A microservices architecture helps teams become autonomous. Teams can make their own decisions about technologies, methodologies, and infrastructure components, for example. However, these choices should respect the formally agreed-upon principles known as shared governance, which express the agreement among teams on how to address the broader strategy for microservices.

Consider these factors:

- Is shared governance in place?
- Do you track decisions and their trade-offs in an architecture journal?
- Can your team easily access your architecture journal?
- Do you have a process for evaluating tools, technologies, and frameworks?

## Assess team composition

You need to have the proper team structure to avoid unnecessary communication across teams. A microservices architecture encourages the formation of small, focused, cross-functional teams and requires a mindset change, which must be preceded by team restructuring.

Consider these factors:

- Are your teams split based on subdomains, following [domain-driven design \(DDD\) principles](#)?
- Are your teams cross-functional, with enough capacity to build and operate related microservices independently?
- How much time is spent in ad hoc activities and tasks that aren't related to projects?
- How much time is spent in cross-team collaboration?
- Do you have a process for identifying and minimizing technical debt?
- How are lessons learned and experience communicated across teams?

## Use the Twelve-Factor methodology

The fundamental goal of choosing a microservices architecture is to deliver value faster and be adaptive to change by following agile practices. The [Twelve-Factor app methodology](#) provides guidelines for building maintainable and scalable applications. These guidelines promote attributes like immutability, ephemeralization, declarative configuration, and automation. By incorporating these guidelines and avoiding common pitfalls, you can create loosely coupled, self-contained microservices.

## Understand the decomposition approach

Transforming a monolithic application to a microservices architecture takes time. Start with edge services. Edge services have fewer dependencies on other services and can be easily decomposed from the system as independent services. We highly recommend patterns like [Strangler Fig](#) and [Anti-corruption Layer](#) to keep the monolithic application in a working state until all services are decomposed into separate microservices. During

segregation, the principles of DDD can help teams choose components or services from the monolithic application based on subdomains.

For example, in an e-commerce system, you might have these modules: cart, product management, order management, pricing, invoice generation, and notification. You decide to start the transformation of the application with the notification module because it doesn't have dependencies on other modules. However, other modules might depend on this module to send out notifications. The notification module can easily be decomposed into a separate microservice, but you'll need to make some changes in the monolithic application to call the new notification service. You decide to transform the invoice generation module next. This module is called after an order is generated. You can use patterns like Strangler and Anti-corruption to support this transformation.

Data synchronization, multi-writes to both monolithic and microservice interfaces, data ownership, schema decomposition, joins, volume of data, and data integrity might make data breakdown and migration difficult. There are several techniques that you can use, like keeping a shared database between microservices, decoupling databases from a group of services based on business capability or domain, and isolating databases from the services. The recommended solution is to decompose each database with each service. In many circumstances, that's not practical. In those cases, you can use patterns like the Database View pattern and the Database Wrapping Service pattern.

## Assess DevOps readiness

When you move to a microservices architecture, it's important to assess your DevOps competence. A microservices architecture is intended to facilitate agile development in applications to increase organizational agility. DevOps is one of the key practices that you should implement to achieve this competence.

When you evaluate your DevOps capability for a microservices architecture, keep these points in mind:

- Do people in your organization know the fundamental practices and principles of DevOps?
- Do teams understand source control tools and their integration with CI/CD pipelines?
- Do you implement DevOps practices properly?
  - Do you follow agile practices?
  - Is continuous integration implemented?
  - Is continuous delivery implemented?
  - Is continuous deployment implemented?

- Is continuous monitoring implemented?
- Is [Infrastructure as Code \(IaC\)](#) in place?
- Do you use the right tools to support CI/CD?
- How is configuration of staging and production environments managed for the application?
- Does the tool chain have community support and a support model and provide proper channels and documentation?

## Identify business areas that change frequently

A microservices architecture is flexible and adaptable. During assessment, drive a discussion in the organization to determine the areas that your colleagues think will change frequently. Building microservices allows teams to quickly respond to changes that customers request and minimize regression testing efforts. In a monolithic application, a change in one module requires numerous levels of regression testing and reduces agility in releasing new versions.

Consider these factors:

- Is the service independently deployable?
- Does the service follow DDD principles?
- Does the service follow [SOLID](#) principles?
- Is the database private to the service?
- Did you build the service by using the supported microservices chassis pattern?

## Assess infrastructure readiness

When you shift to a microservices architecture, infrastructure readiness is a critical point to consider. The application's performance, availability, and scalability will be affected if the infrastructure isn't properly set up or if the right services or components aren't used. Sometimes an application is created with all the suggested methodologies and procedures, but the infrastructure is inadequate. This results in poor performance and maintenance.

Consider these factors when you evaluate your infrastructure readiness:

- Does the infrastructure ensure the scalability of the services deployed?
- Does the infrastructure support provisioning through scripts that can be automated via CI/CD?
- Does the deployment infrastructure offer an SLA for availability?
- Do you have a disaster recovery (DR) plan and routine drill schedules?

- Is the data replicated to different regions for DR?
- Do you have a data backup plan?
- Are the deployment options documented?
- Is the deployment infrastructure monitored?
- Does the deployment infrastructure support self-healing of services?

## Assess release cycles

Microservices are adaptive to change and take advantage of agile development to shorten release cycles and bring value to customers more. Consider these factors when you evaluate your release cycles:

- How often do you build and release applications?
- How often do your releases fail after deployment?
- How long does it take to recover or remediate problems after an outage?
- Do you use semantic versioning for your applications?
- Do you maintain different environments and propagate the same release in a sequence (for example, first to staging and then to production)?
- Do you use versioning for your APIs?
- Do you follow proper versioning guidelines for APIs?
- When do you change an API version?
- What's your approach for handling API versioning?
  - URI path versioning
  - Query parameter versioning
  - Content-type versioning
  - Custom header versioning
- Do you have a practice in place for event versioning?

## Assess communication across services

Microservices are self-contained services that communicate with each other across process boundaries to address business scenarios. To get reliable and dependable communication, you need to select an appropriate communication protocol.

Take these factors into consideration:

- Are you following an API-first approach, where APIs are treated as first-class citizens?
- Do you have long-chain operations, where multiple services communicate in sequence over a synchronous communication protocol?
- Have you considered asynchronous communication anywhere in the system?

- Have you assessed the message broker technology and its capabilities?
- Do you understand the throughput of messages that services receive and process?
- Do you use direct client-to-service communication?
- Do you need to persist messages at the message broker level?
- Are you using the [Materialized View pattern](#) to address the chatty behavior of microservices?
- Have you implemented Retry, Circuit Breaker, Exponential Backoff, and Jitter for reliable communication? A common way to handle this is to use the [Ambassador pattern](#).
- Do you have defined domain events to facilitate communication between microservices?

## Evaluate how services are exposed to clients

An API gateway is one of the core components in a microservices architecture. Directly exposing services to the clients creates problems in terms of manageability, control, and dependable communication. An API gateway serves as a proxy server, intercepting traffic and routing it to back-end services. If you need to filter traffic by IP address, authorization, mock responses, and so on, you can do it at the API gateway level without making any changes to the services.

Take these factors into consideration:

- Are the services directly consumed by clients?
- Is there an API gateway that acts as a facade for all the services?
- Can the API gateway set up policies like quota limits, mocking responses, and filtering of IP addresses?
- Are you using multiple API gateways to address the needs of various types of clients, like mobile apps, web apps, and partners?
- Does your API gateway provide a portal where clients can discover and subscribe to services, like a developer portal in [Azure API Management](#)?
- Does your solution provide L7 load balancing or Web Application Firewall (WAF) capabilities along with the API gateway?

## Assess transaction handling

Distributed transactions facilitate the execution of multiple operations as a single unit of work. In a microservices architecture, the system is decomposed into numerous services. A single business use case is addressed by multiple microservices as part of a single distributed transaction. In a distributed transaction, a command is an operation that starts when an event occurs. The event triggers the an operation in the system. If the

operation succeeds, it might trigger another command, which can then trigger another event, and so on until all the transactions are completed or rolled back, depending on whether the transaction succeeds.

Take the following considerations into account:

- How many distributed transactions are there in the system?
- What's your approach to handling distributed transactions? Evaluate the use of the [Saga pattern](#) with orchestration or choreography.
- How many transactions span multiple services?
- Are you following an ACID or BASE transaction model to achieve data consistency and integrity?
- Are you using long-chaining operations for transactions that span multiple services?

## Assess your service development model

One of the greatest benefits of microservices architectures is technology diversity. Microservices-based systems enable teams to develop services by using the technologies that they choose. In traditional application development, you might reuse existing code when you build new components, or create an internal development framework to reduce development effort. The use of multiple technologies can prevent code reuse.

Consider these factors:

- Do you use a service template to kickstart new service development?
- Do you follow the Twelve-Factor app methodology and use a single code base for microservices, isolating dependencies and externalizing configuration?
- Do you keep sensitive configuration in key vaults?
- Do you containerize your services?
- Do you know your data consistency requirements?

## Assess your deployment approach

Your deployment approach is your method for releasing versions of your application across various deployment environments. Microservices-based systems provide the agility to release versions faster than you can with traditional applications.

When you assess your deployment plan, consider these factors:

- Do you have a strategy for deploying your services?

- Are you using modern tools and technologies to deploy your services?
- What kind of collaboration is required among teams when you deploy services?
- Do you provision infrastructure by using Infrastructure as Code (IaC)?
- Do you use DevOps capabilities to automate deployments?
- Do you propagate the same builds to multiple environments, as suggested by the Twelve-Factor app methodology?

## Assess your hosting platform

Scalability is one of the key advantages of microservices architectures. That's because microservices are mapped to business domains, so each service can be scaled independently. A monolithic application is deployed as a single unit on a hosting platform and needs to be scaled holistically. That results in downtime, deployment risk, and maintenance. Your monolithic application might be well designed, with components addressing individual business domains. But because of a lack of process boundaries, the potential for violating the principles of single responsibility becomes more difficult. This eventually results in spaghetti code. Because the application is composed and deployed as a single hosting process, scalability is difficult.

Microservices enables teams to choose the right hosting platform to support their scalability needs. Various hosting platforms are available to address these challenges by providing capabilities like autoscaling, elastic provisioning, higher availability, faster deployment, and easy monitoring.

Consider these factors:

- What hosting platform do you use to deploy your services (virtual machines, containers, serverless)?
- Is the hosting platform scalable? Does it support autoscaling?
- How long does it take to scale your hosting platform?
- Do you understand the SLAs that various hosting platforms provide?
- Does your hosting platform support disaster recovery?

## Assess services monitoring

Monitoring is an important element of a microservices-based application. Because the application is divided into a number of services that run independently, troubleshooting errors can be daunting. If you use proper semantics to monitor your services, your teams can easily monitor, investigate, and respond to errors.

Consider these factors:

- Do you monitor your deployed services?
- Do you have a logging mechanism? What tools do you use?
- Do you have a distributed tracing infrastructure in place?
- Do you record exceptions?
- Do you maintain business error codes for quick identification of problems?
- Do you use health probes for services?
- Do you use semantic logging? Have you implemented key metrics, thresholds, and indicators?
- Do you mask sensitive data during logging?

## Assess correlation token assignment

In a microservices architecture, an application is composed of various microservices that are hosted independently, interacting with each other to serve various business use cases. When an application interacts with back-end services to perform an operation, you can assign a unique number, known as a correlation token, to the request. We recommend that you consider using correlation tokens, because they can help you troubleshoot errors. They help you determine the root cause of a problem, assess the impact, and determine an approach to remediate the problem.

You can use correlation tokens to retrieve the request trail by identifying which services contain the correlation token and which don't. The services that don't have the correlation token for that request failed. If a failure occurs, you can later retry the transaction. To enforce idempotency, only services that don't have the correlation token will serve the request.

For example, if you have a long chain of operations that involves many services, passing a correlation token to services can help you investigate issues easily if any of the services fails during a transaction. Each service usually has its own database. The correlation token is kept in the database record. In case of a transaction replay, services that have that particular correlation token in their databases ignore the request. Only services that don't have the token serve the request.

Consider these factors:

- At which stage do you assign the correlation token?
- Which component assigns the correlation token?
- Do you save correlation tokens in the service's database?
- What's the format of correlation tokens?
- Do you log correlation tokens and other request information?

# Evaluate the need for a microservices chassis framework

A microservices chassis framework is a base framework that provides capabilities for cross-cutting concerns like logging, exception handling, distributed tracing, security, and communication. When you use a chassis framework, you focus more on implementing the service boundary than interacting with infrastructure functionality.

For example, say you're building a cart management service. You want to validate the incoming token, write logs to the logging database, and communicate with another service by invoking that service's endpoint. If you use a microservices chassis framework, you can reduce development efforts. Dapr is one system that provides various building blocks for implementing cross-cutting concerns.

Consider these factors:

- Do you use a microservices chassis framework?
- Do you use Dapr to interact with cross-cutting concerns?
- Is your chassis framework language agnostic?
- Is your chassis framework generic, so it supports all kinds of applications? A chassis framework shouldn't contain application-specific logic.
- Does your chassis framework provide a mechanism to use the selected components or services as needed?

# Assess your approach to application testing

Traditionally, testing is done after development is completed and the application is ready to roll out to user acceptance testing (UAT) and production environments. There's currently a shift in this approach, moving the testing early (left) in the application development lifecycle. Shift-left testing increases the quality of applications because testing is done during each phase of the application development lifecycle, including the design, development, and post-development phases.

For example, when you build an application, you start by designing an architecture. When you use the shift-left approach, you test the design for vulnerabilities by using tools like [Microsoft Threat Modeling](#). When you start development, you can scan your source code by running tools like static application security testing (SAST) and using other analyzers to uncover problems. After you deploy the application, you can use tools like dynamic application security testing (DAST) to test it while it's hosted. Functional testing, chaos testing, penetration testing, and other kinds of testing happen later.

Consider these factors:

- Do you write test plans that cover the entire development lifecycle?
- Do you include testers in requirements meetings and in the entire application development lifecycle?
- Do you use test-driven design or behavior-driven design?
- Do you test user stories? Do you include acceptance criteria in your user stories?
- Do you test your design by using tools like Microsoft Threat Modeling?
- Do you write unit tests?
- Do you use static code analyzers or other tools to measure code quality?
- Do you use automated tools to test applications?
- Do you implement [Secure DevOps](#) practices?
- Do you do integration testing, end-to-end application testing, load/performance testing, penetration testing, and chaos testing?

## Assess microservices security

Service protection, secure access, and secure communication are among the most important considerations for a microservices architecture. For example, a microservices-based system that spans multiple services and uses token validation for each service isn't a viable solution. This system would affect the agility of the overall system and the potential of introducing implementation drift across services.

Security concerns are usually handled by the API gateway and the application firewall. The gateway and firewall take incoming requests, validate tokens, and apply various filters and policies, like implementing OWASP Top 10 principles to intercept traffic. Finally, they send the request to the back-end microservices. This configuration helps developers focus on business needs rather than the cross-cutting concern of security.

Consider these factors:

- Are authentication and authorization required for the service?
- Are you using an API gateway to validate tokens and incoming requests?
- Are you using SSL or mTLS to provide security for service communication?
- Do you have network security policies in place to allow the required communication among services?
- Are you using firewalls (L4, L7) where applicable to provide security for internal and external communications?
- Do you use security policies in API Gateway to control traffic?

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal authors:

- [Ovais Mehboob Ahmed Khan](#) | Senior Cloud Solution Architect - Engineering
- [Nabil Siddiqui](#) | Cloud Solution Architect - Digital & Application Innovation

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [Microservices on Azure](#)
- [Book: Embrace Microservices Design](#)
- [Introduction to deployment patterns](#)
- [Design a microservices-oriented application](#)

## Related resources

- [Microservices architecture style](#)
- [Build microservices on Azure](#)
- [Microservices architecture on Azure Kubernetes Service](#)
- [Using domain analysis to model microservices](#)
- [Using tactical DDD to design microservices](#)
- [Design a microservices architecture](#)

# Compare Java application hosting options on Azure

Azure Spring Apps

Azure App Service

Azure Kubernetes Service (AKS)

Azure Virtual Machines

Azure offers many options for teams to build and deploy Java applications. This article covers mainstream scenarios for Java on Azure and provides high-level planning suggestions and considerations.

*Apache®, Apache Kafka®, Apache Struts®, Apache Tomcat®, and the flame logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.*

## Platform

Before you select a cloud scenario for your Java application, identify its platform. Most Java applications use one of the following platforms:

- [Spring Boot JAR applications](#)
- [Spring Cloud applications](#)
- [Web applications](#)
- [Jakarta EE applications](#)

## Spring Boot JAR applications

Spring Boot JAR applications are typically invoked directly from the command line. They handle web requests. Instead of relying on an application server to handle HTTP requests, these applications incorporate HTTP communication and other dependencies directly into the application package. Such applications are often built with frameworks such as [Spring Boot](#), [Dropwizard](#), [Micronaut](#), [MicroProfile](#), and [Vert.x](#).

These applications are packaged into archives that have the *.jar* extension, known as JAR files.

## Spring Cloud applications

The *microservice architectural style* is an approach to developing a single application as a suite of small services. Each service runs in its own process and communicates by using lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities.

Automated deployment machinery independently deploys these microservices. There's a minimum of centralized management, which might be written in different programming languages and use different data storage technologies. Such services are often built with frameworks such as [Spring Cloud](#).

These services are packaged into multiple applications as JAR files.

## Web applications

Web applications run inside a servlet container. Some use servlet APIs directly, while others use other frameworks that encapsulate servlet APIs, such as [Apache Struts](#), [Spring MVC](#), and [JavaServer Faces](#).

Web applications are packaged into archives that have the *.war* extension, known as WAR files.

## Jakarta EE applications

Jakarta Enterprise Edition (Jakarta EE) applications can contain some, all, or none of the elements of web applications. They can also contain and consume many more components, as defined by the Jakarta EE specification. Jakarta EE applications were formerly known as *Java EE applications* or *J2EE applications*.

Jakarta EE applications can be packaged as WAR files or as archives that have the *.ear* extension, known as EAR files.

Jakarta EE applications must be deployed onto application servers that are Jakarta EE compliant. Examples include [WebLogic](#), [WebSphere](#), [WildFly](#), [GlassFish](#), and [Payara](#).

Applications that rely only on features provided by the Jakarta EE specification can be migrated from one compliant application server onto another. If your application is dependent on a specific application server, you might need to select an Azure service destination that permits you to host that application server.

## Platform options

Use the following table to identify potential platforms for your application type.

 Expand table

	Azure Spring Apps	App Service Java SE	App Service Tomcat	App Service JBoss EAP	Azure Container Apps	AKS	Virtual Machines
<b>Spring Boot / JAR applications</b>	✓	✓		✓	✓	✓	✓
<b>Spring Cloud applications</b>		✓			✓	✓	✓
<b>Web applications</b>	✓		✓	✓	✓	✓	✓
<b>Jakarta EE applications</b>				✓		✓	✓
<b>Azure region availability</b>	<a href="#">Details</a>						

Azure Kubernetes Service (AKS) and Virtual Machines support all application types, but they require that your team to take on more responsibilities, as described in the next section.

## Supportability

Besides the platform choices, modern Java applications might have other supportability needs, such as:

- [Batch or scheduled jobs](#)
- [Virtual network integration](#)
- [Serverless](#)
- [Containerization](#)

## Batch or scheduled jobs

Instead of waiting for requests or user input, some applications run briefly, run a particular workload, and then exit. Sometimes, such jobs need to run once or at regular, scheduled intervals. On-premises, such jobs are often invoked from a server's cron table.

These applications are packaged as JAR files.

### ⓘ Note

If your application uses a scheduler, such as Spring Batch or Quartz, to run scheduled tasks, we strongly recommend that you run those tasks outside of the application. If your application scales to multiple instances in the cloud, the same job can run more than once. If your scheduling mechanism uses the host's local time zone, there might be undesired behavior when you scale an application across regions.

## Virtual network integration

When you deploy a Java application in your virtual network, it has outbound dependencies on services outside of the virtual network. For management and operations, your project must have access to certain ports and fully qualified domain names. With Azure Virtual Networks, you can place many of your Azure resources in a non-internet routable network. The *virtual network integration* feature enables your applications to access resources in or through a virtual network. Virtual network integration doesn't enable your applications to be accessed privately.

## Serverless development model

Serverless is a cloud-native development model that allows developers to build and run applications without having to manage servers. With serverless applications, the cloud service provider automatically provisions, scales, and manages the infrastructure required to run the code. Servers still exist in the serverless model. They're abstracted away from application development.

## Containerization

Containerization is the packaging together of software code with all its necessary components, like libraries, frameworks, and other dependencies. The application is isolated in its own container.

## CI/CD

Continuous integration and continuous delivery (CI/CD) is a method to frequently deliver applications to customers by introducing automation into the stages of application development. The main concepts in CI/CD are *continuous integration*, *continuous delivery*, and *continuous deployment*. All of the Azure choices support most CI/CD tooling. For example, you might use solutions such as [Azure Pipelines](#) or [Jenkins](#).

## Open-source search engine

Searches are integral parts of any application. If speed, performance, and high availability are critical, searches on terabytes and petabytes of data can be challenging. When you host Java applications on Azure, plan to host your related Solr and Elasticsearch instances. Alternatively, consider migrating to [Azure Cognitive Search](#).

## Big data tools

Big data tools enable the automation of data flow among the software systems. They support scalable, robust, and streamlined data routing graphs along with system mediation logic. They're utilized to build live data flow pipelines and stream applications. Learn how [Nifi](#) and [Apache Kafka](#) on Azure might be suitable for your needs.

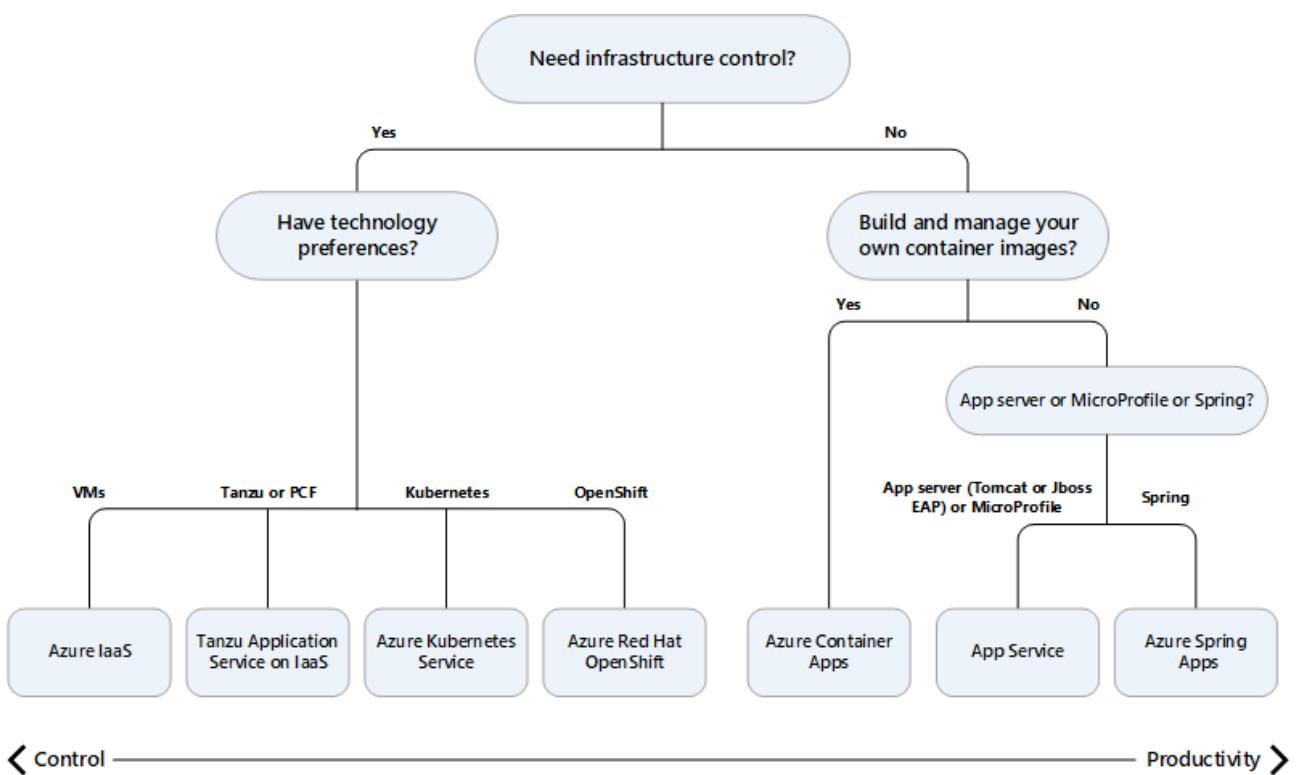
## Supportability options

Use the following table to identify potential options for your application type. AKS and Virtual Machines support all application types, but they require your team to take on more responsibilities.

 [Expand table](#)

	Azure Spring Apps	App Service Java SE	App Service Tomcat	App Service JBoss EAP	Azure Container Apps	AKS	Virtual Machines
<b>Batch or scheduled jobs</b>	✓				✓	✓	✓
<b>Virtual network integration</b>	✓	✓	✓	✓	✓	✓	✓
<b>Serverless</b>	✓				✓	✓	✓
<b>Containerization</b>	✓	✓	✓	✓	✓	✓	✓
<b>Azure region availability</b>	<a href="#">Details ↗</a>						

Also, refer to this decision tree.



Download a [Visio file](#) of this diagram.

## Build or migrate Java applications

To build or migrate the Java applications, identify the Java platform of your applications. Some popular platforms are [Java SE](#), [Jakarta EE](#), and [MicroProfile](#).

### Java SE

Java Platform, Standard Edition (Java SE) is a computing platform for the development and deployment of portable code for desktop and server environments. Popular projects built on Java SE include Spring Boot, Spring Cloud, [Spring Framework](#), and [Apache Tomcat](#).

### Jakarta EE

Jakarta EE is the open source future of cloud-native enterprise Java. It's a set of specifications that extend Java SE with enterprise features such as distributed computing and web services. Jakarta EE applications run on reference runtimes. These runtimes can be microservices or application servers. They handle transactions, security, scalability, concurrency, and management of the components the application deploys.

### MicroProfile

The MicroProfile project provides a collection of specifications designed to help developers build Enterprise Java cloud-native microservices. [Quarkus](#) and [Open Liberty](#) are popular implementations of MicroProfile.

## Build or migrate summary

The following table provides build or migration information by application type and Azure service.

	Type	Java SE	MicroProfile	JarkartaSE
<b>Virtual Machine</b>	IaaS	✓	✓	✓
<b>VMware Tanzu</b>	IaaS	✓		
<b>Azure Kubernetes Service</b>	Container	✓	✓	✓
<b>Red Hat OpenShift</b>	Container	✓	✓	✓
<b>Azure Container App</b>	PaaS	✓	✓	
<b>JBoss EAP</b>	PaaS App Service	✓		✓
<b>Apache Tomcat</b>	PaaS App Service	✓		
<b>Java SE</b>	PaaS App Service	✓	✓	
<b>Azure Spring Apps</b>	PaaS	✓		

## Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- [Asir Vedamuthu Selvasingh](#) | Principal Program Manager
- [Hang Wang](#) | Product Manager
- [Xinyi Zhang](#) | Principal PM Manager

To see non-public LinkedIn profiles, sign in to LinkedIn.

## Next steps

- [Azure Container Apps overview](#)
- [Azure Kubernetes Service](#)
- [Azure Spring Apps documentation](#)
- [Azure Virtual network integration](#)
- [Virtual machines in Azure](#)

## Related resources

- [Expose Azure Spring Apps through a reverse proxy](#)
- [Java CI/CD using Jenkins and Azure Web Apps](#)
- [Microservices architecture design](#)
- [Multicloud solutions with the Serverless Framework](#)

- Virtual network integrated serverless microservices

# Using domain analysis to model microservices

Article • 12/16/2022

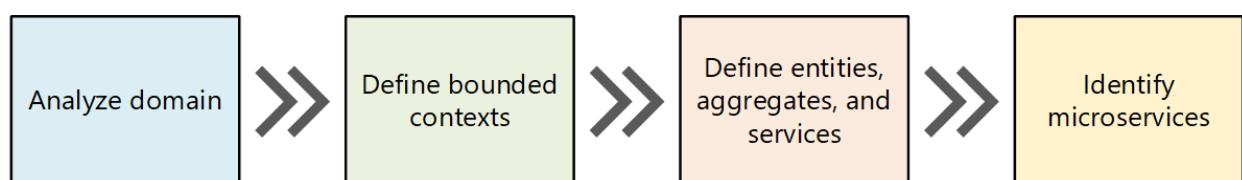
One of the biggest challenges of microservices is to define the boundaries of individual services. The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought. There is no mechanical process that will produce the "right" design. You have to think deeply about your business domain, requirements, and goals. Otherwise, you can end up with a haphazard design that exhibits some undesirable characteristics, such as hidden dependencies between services, tight coupling, or poorly designed interfaces. This article shows a domain-driven approach to designing microservices.

This article uses a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation [here](#).

## Introduction

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion. Microservices are *loosely coupled* if you can change one service without requiring other services to be updated at the same time. A microservice is *cohesive* if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history. A service should encapsulate domain knowledge and abstract that knowledge from clients. For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices. DDD has two distinct phases, strategic and tactical. In strategic DDD, you are defining the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities. Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services. These tactical patterns will help you to design microservices that are both loosely coupled and cohesive.



In this article and the next, we'll walk through the following steps, applying them to the Drone Delivery application:

1. Start by analyzing the business domain to understand the application's functional requirements. The output of this step is an informal description of the domain, which can be refined into a more formal set of domain models.
2. Next, define the *bounded contexts* of the domain. Each bounded context contains a domain model that represents a particular subdomain of the larger application.
3. Within a bounded context, apply tactical DDD patterns to define entities, aggregates, and domain services.
4. Use the results from the previous step to identify the microservices in your application.

In this article, we cover the first three steps, which are primarily concerned with DDD. In the next article, we'll identify the microservices. However, it's important to remember that DDD is an iterative, ongoing process. Service boundaries aren't fixed in stone. As an application evolves, you may decide to break apart a service into several smaller services.

#### Note

This article doesn't show a complete and comprehensive domain analysis. We deliberately kept the example brief, to illustrate the main points. For more background on DDD, we recommend Eric Evans' *Domain-Driven Design*, the book that first introduced the term. Another good reference is *Implementing Domain-Driven Design* by Vaughn Vernon.

## Scenario: Drone delivery

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then

iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

## Analyze the domain

Using a DDD approach will help you to design microservices so that every service forms a natural fit to a functional business requirement. It can help you to avoid the trap of letting organizational boundaries or technology choices dictate your design.

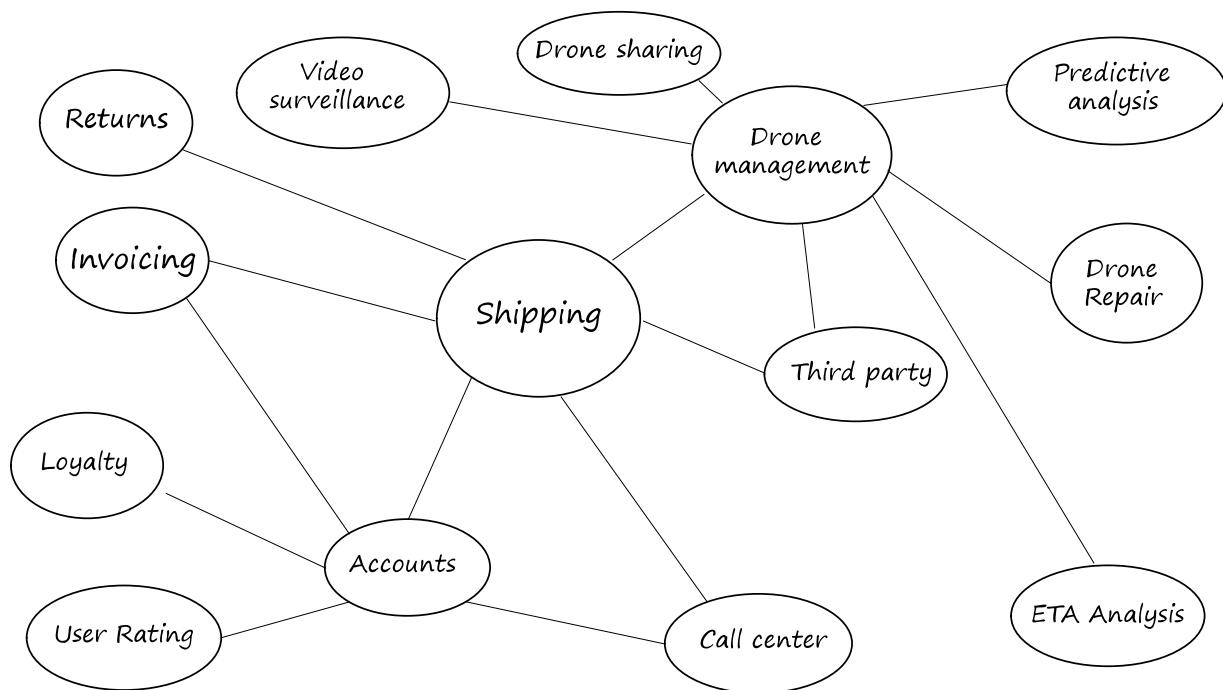
Before writing any code, you need a bird's eye view of the system that you are creating. DDD starts by modeling the business domain and creating a *domain model*. The domain model is an abstract model of the business domain. It distills and organizes domain knowledge, and provides a common language for developers and domain experts.

Start by mapping all of the business functions and their connections. This will likely be a collaborative effort that involves domain experts, software architects, and other stakeholders. You don't need to use any particular formalism. Sketch a diagram or draw on whiteboard.

As you fill in the diagram, you may start to identify discrete subdomains. Which functions are closely related? Which functions are core to the business, and which provide ancillary services? What is the dependency graph? During this initial phase, you aren't concerned with technologies or implementation details. That said, you should note the place where the application will need to integrate with external systems, such as CRM, payment processing, or billing systems.

## Example: Drone delivery application

After some initial domain analysis, the Fabrikam team came up with a rough sketch that depicts the Drone Delivery domain.



- **Shipping** is placed in the center of the diagram, because it's core to the business. Everything else in the diagram exists to enable this functionality.
- **Drone management** is also core to the business. Functionality that is closely related to drone management includes **drone repair** and using **predictive analysis** to predict when drones need servicing and maintenance.
- **ETA analysis** provides time estimates for pickup and delivery.
- **Third-party transportation** will enable the application to schedule alternative transportation methods if a package cannot be shipped entirely by drone.
- **Drone sharing** is a possible extension of the core business. The company may have excess drone capacity during certain hours, and could rent out drones that would otherwise be idle. This feature will not be in the initial release.
- **Video surveillance** is another area that the company might expand into later.
- **User accounts**, **Invoicing**, and **Call center** are subdomains that support the core business.

Notice that at this point in the process, we haven't made any decisions about implementation or technologies. Some of the subsystems may involve external software systems or third-party services. Even so, the application needs to interact with these systems and services, so it's important to include them in the domain model.

### ① Note

When an application depends on an external system, there is a risk that the external system's data schema or API will leak into your application, ultimately compromising the architectural design. This is particularly true with legacy systems that may not follow modern best practices, and may use convoluted data schemas or obsolete APIs. In that case, it's important to have a well-defined boundary

between these external systems and the application. Consider using the **Strangler Fig pattern** or the **Anti-Corruption Layer pattern** for this purpose.

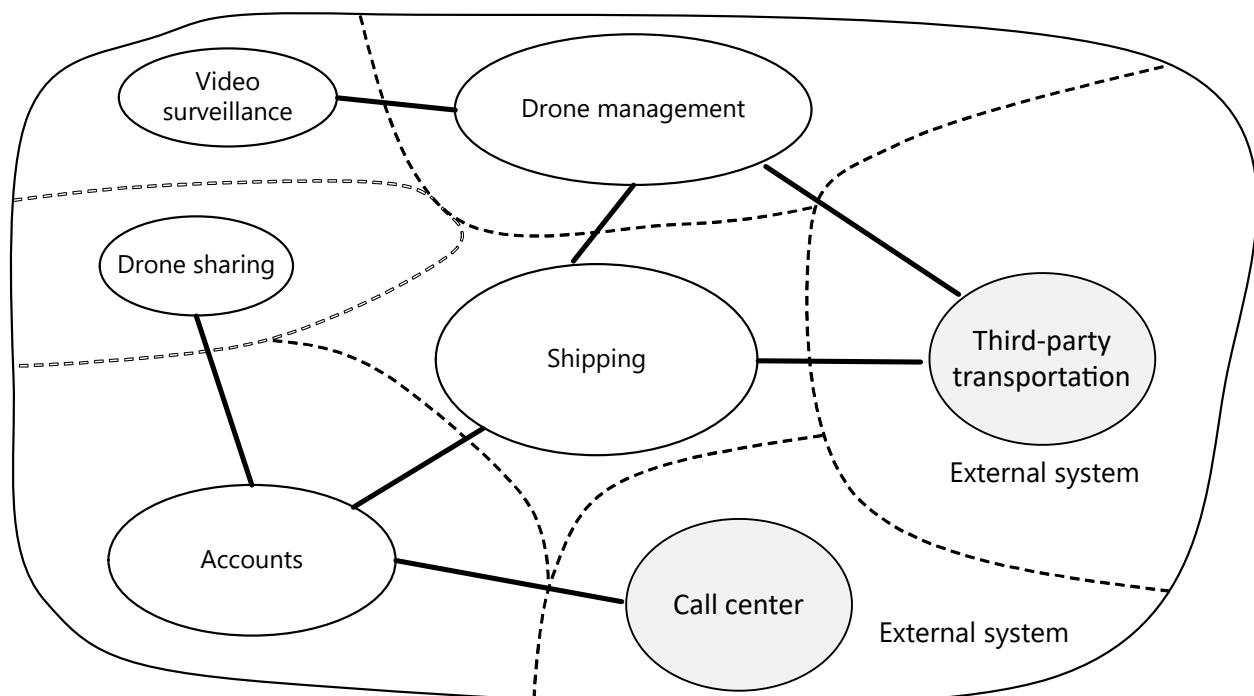
## Define bounded contexts

The domain model will include representations of real things in the world — users, drones, packages, and so forth. But that doesn't mean that every part of the system needs to use the same representations for the same things.

For example, subsystems that handle drone repair and predictive analysis will need to represent many physical characteristics of drones, such as their maintenance history, mileage, age, model number, performance characteristics, and so on. But when it's time to schedule a delivery, we don't care about those things. The scheduling subsystem only needs to know whether a drone is available, and the ETA for pickup and delivery.

If we tried to create a single model for both of these subsystems, it would be unnecessarily complex. It would also become harder for the model to evolve over time, because any changes will need to satisfy multiple teams working on separate subsystems. Therefore, it's often better to design separate models that represent the same real-world entity (in this case, a drone) in two different contexts. Each model contains only the features and attributes that are relevant within its particular context.

This is where the DDD concept of *bounded contexts* comes into play. A bounded context is simply the boundary within a domain where a particular domain model applies. Looking at the previous diagram, we can group functionality according to whether various functions will share a single domain model.



Bounded contexts are not necessarily isolated from one another. In this diagram, the solid lines connecting the bounded contexts represent places where two bounded contexts interact. For example, Shipping depends on User Accounts to get information about customers, and on Drone Management to schedule drones from the fleet.

In the book *Domain Driven Design*, Eric Evans describes several patterns for maintaining the integrity of a domain model when it interacts with another bounded context. One of the main principles of microservices is that services communicate through well-defined APIs. This approach corresponds to two patterns that Evans calls Open Host Service and Published Language. The idea of Open Host Service is that a subsystem defines a formal protocol (API) for other subsystems to communicate with it. Published Language extends this idea by publishing the API in a form that other teams can use to write clients. In the article [Designing APIs for microservices](#), we discuss using [OpenAPI Specification](#) (formerly known as Swagger) to define language-agnostic interface descriptions for REST APIs, expressed in JSON or YAML format.

For the rest of this journey, we will focus on the Shipping bounded context.

## Next steps

After completing a domain analysis, the next step is to apply tactical DDD, to define your domain models with more precision.

Tactical DDD

## Related resources

- [Microservices architecture design](#)
- [Design a microservices architecture](#)
- [Identify microservice boundaries](#)
- [Choose an Azure compute option for microservices](#)

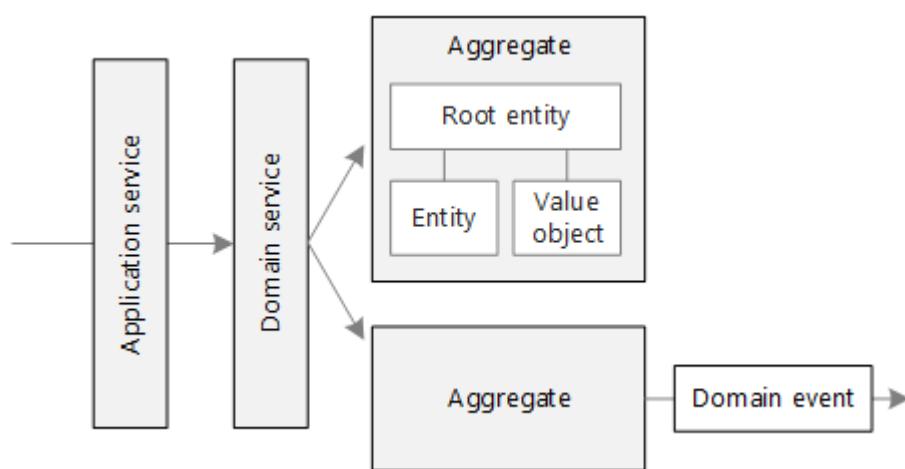
# Using tactical DDD to design microservices

Azure Migrate

During the strategic phase of domain-driven design (DDD), you are mapping out the business domain and defining bounded contexts for your domain models. Tactical DDD is when you define your domain models with more precision. The tactical patterns are applied within a single bounded context. In a microservices architecture, we are particularly interested in the entity and aggregate patterns. Applying these patterns will help us to identify natural boundaries for the services in our application (see the [next article](#) in this series). As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context. First, we'll review the tactical patterns. Then we'll apply them to the Shipping bounded context in the Drone Delivery application.

## Overview of the tactical patterns

This section provides a brief summary of the tactical DDD patterns, so if you are already familiar with DDD, you can probably skip this section. The patterns are described in more detail in chapters 5 – 6 of Eric Evans' book, and in *Implementing Domain-Driven Design* by Vaughn Vernon.



**Entities.** An entity is an object with a unique identity that persists over time. For example, in a banking application, customers and accounts would be entities.

- An entity has a unique identifier in the system, which can be used to look up or retrieve the entity. That doesn't mean the identifier is always exposed directly to users. It could be a GUID or a primary key in a database.

- An identity may span multiple bounded contexts, and may endure beyond the lifetime of the application. For example, bank account numbers or government-issued IDs are not tied to the lifetime of a particular application.
- The attributes of an entity may change over time. For example, a person's name or address might change, but they are still the same person.
- An entity can hold references to other entities.

**Value objects.** A value object has no identity. It is defined only by the values of its attributes. Value objects are also immutable. To update a value object, you always create a new instance to replace the old one. Value objects can have methods that encapsulate domain logic, but those methods should have no side-effects on the object's state.

Typical examples of value objects include colors, dates and times, and currency values.

**Aggregates.** An aggregate defines a consistency boundary around one or more entities. Exactly one entity in an aggregate is the root. Lookup is done using the root entity's identifier. Any other entities in the aggregate are children of the root, and are referenced by following pointers from the root.

The purpose of an aggregate is to model transactional invariants. Things in the real world have complex webs of relationships. Customers create orders, orders contain products, products have suppliers, and so on. If the application modifies several related objects, how does it guarantee consistency? How do we keep track of invariants and enforce them?

Traditional applications have often used database transactions to enforce consistency. In a distributed application, however, that's often not feasible. A single business transaction may span multiple data stores, or may be long running, or may involve third-party services. Ultimately it's up to the application, not the data layer, to enforce the invariants required for the domain. That's what aggregates are meant to model.

### ① Note

An aggregate might consist of a single entity, without child entities. What makes it an aggregate is the transactional boundary.

**Domain and application services.** In DDD terminology, a service is an object that implements some logic without holding any state. Evans distinguishes between *domain services*, which encapsulate domain logic, and *application services*, which provide technical functionality, such as user authentication or sending an SMS message. Domain services are often used to model behavior that spans multiple entities.

## (!) Note

The term *service* is overloaded in software development. The definition here is not directly related to microservices.

**Domain events.** Domain events can be used to notify other parts of the system when something happens. As the name suggests, domain events should mean something within the domain. For example, "a record was inserted into a table" is not a domain event. "A delivery was cancelled" is a domain event. Domain events are especially relevant in a microservices architecture. Because microservices are distributed and don't share data stores, domain events provide a way for microservices to coordinate with each other. The article [Interservice communication](#) discusses asynchronous messaging in more detail.

There are a few other DDD patterns not listed here, including factories, repositories, and modules. These can be useful patterns for when you are implementing a microservice, but they are less relevant when designing the boundaries between microservice.

## Drone delivery: Applying the patterns

We start with the scenarios that the Shipping bounded context must handle.

- A customer can request a drone to pick up goods from a business that is registered with the drone delivery service.
- The sender generates a tag (barcode or RFID) to put on the package.
- A drone will pick up and deliver a package from the source location to the destination location.
- When a customer schedules a delivery, the system provides an ETA based on route information, weather conditions, and historical data.
- When the drone is in flight, a user can track the current location and the latest ETA.
- Until a drone has picked up the package, the customer can cancel a delivery.
- The customer is notified when the delivery is completed.
- The sender can request delivery confirmation from the customer, in the form of a signature or finger print.
- Users can look up the history of a completed delivery.

From these scenarios, the development team identified the following **entities**.

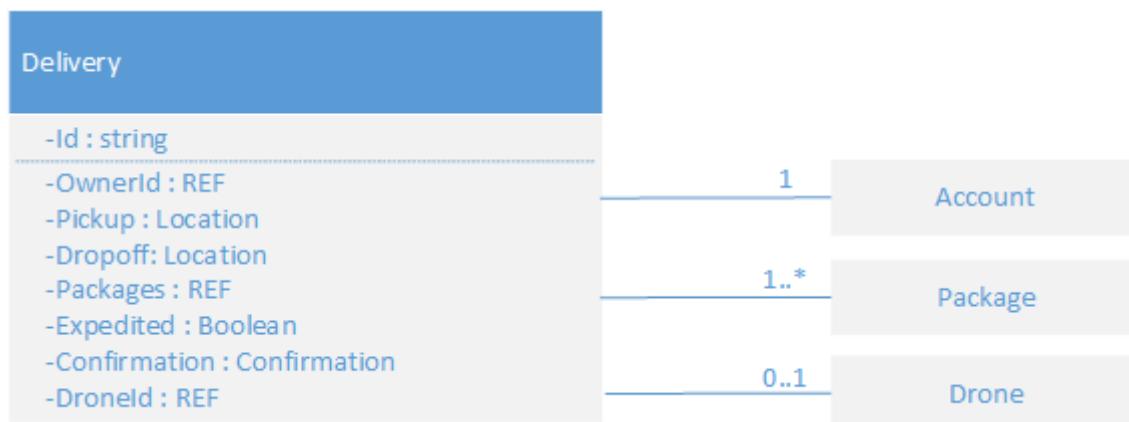
- Delivery
- Package
- Drone

- Account
- Confirmation
- Notification
- Tag

The first four, Delivery, Package, Drone, and Account, are all **aggregates** that represent transactional consistency boundaries. Confirmations and Notifications are child entities of Deliveries, and Tags are child entities of Packages.

The **value objects** in this design include Location, ETA, PackageWeight, and PackageSize.

To illustrate, here is a UML diagram of the Delivery aggregate. Notice that it holds references to other aggregates, including Account, Package, and Drone.

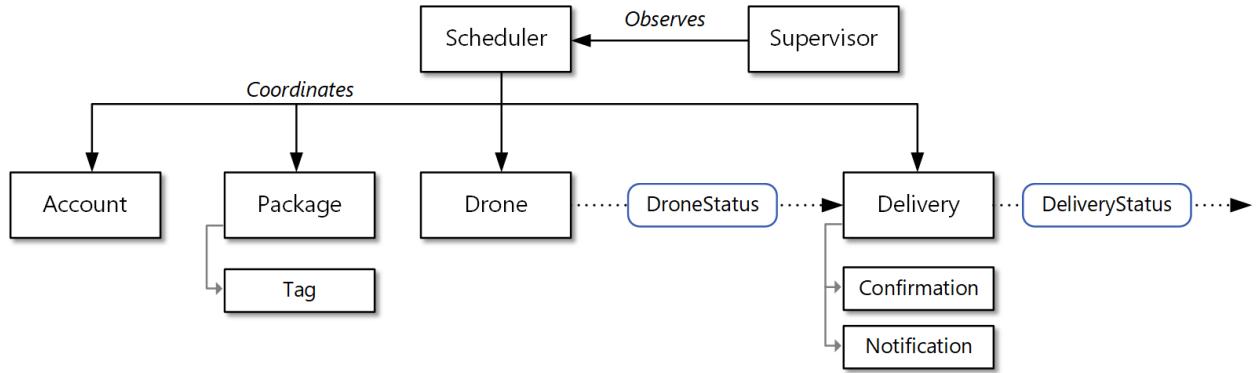


There are two domain events:

- While a drone is in flight, the Drone entity sends DroneStatus events that describe the drone's location and status (in-flight, landed).
- The Delivery entity sends DeliveryTracking events whenever the stage of a delivery changes. These include DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff, and DeliveryCompleted.

Notice that these events describe things that are meaningful within the domain model. They describe something about the domain, and aren't tied to a particular programming language construct.

The development team identified one more area of functionality, which doesn't fit neatly into any of the entities described so far. Some part of the system must coordinate all of the steps involved in scheduling or updating a delivery. Therefore, the development team added two **domain services** to the design: a *Scheduler* that coordinates the steps, and a *Supervisor* that monitors the status of each step, in order to detect whether any steps have failed or timed out. This is a variation of the [Scheduler Agent Supervisor pattern](#).



## Next steps

The next step is to define the boundaries for each microservice.

[Identify microservice boundaries](#)

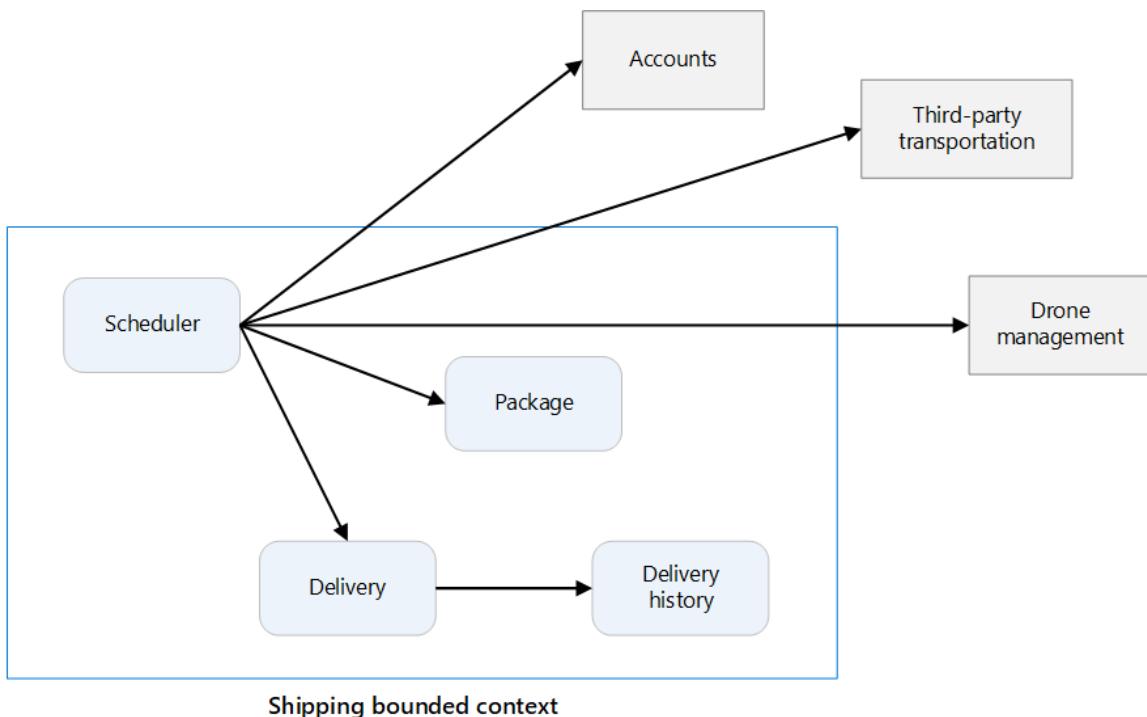
## Related resources

- [Microservices architecture design](#)
- [Design a microservices architecture](#)
- [Using domain analysis to model microservices](#)
- [Choose an Azure compute option for microservices](#)

# Identify microservice boundaries

Azure DevOps

What is the right size for a microservice? You often hear something to the effect of, "not too big and not too small" — and while that's certainly correct, it's not very helpful in practice. But if you start from a carefully designed domain model, it's much easier to reason about microservices.



This article uses a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation [here](#).

## From domain model to microservices

In the [previous article](#), we defined a set of bounded contexts for a Drone Delivery application. Then we looked more closely at one of these bounded contexts, the Shipping bounded context, and identified a set of entities, aggregates, and domain services for that bounded context.

Now we're ready to go from domain model to application design. Here's an approach that you can use to derive microservices from the domain model.

1. Start with a bounded context. In general, the functionality in a microservice should not span more than one bounded context. By definition, a bounded context marks the boundary of a particular domain model. If you find that a microservice mixes different domain models together, that's a sign that you may need to go back and refine your domain analysis.
2. Next, look at the aggregates in your domain model. Aggregates are often good candidates for microservices. A well-designed aggregate exhibits many of the characteristics of a well-designed microservice, such as:
  - An aggregate is derived from business requirements, rather than technical concerns such as data access or messaging.
  - An aggregate should have high functional cohesion.
  - An aggregate is a boundary of persistence.
  - Aggregates should be loosely coupled.
3. Domain services are also good candidates for microservices. Domain services are stateless operations across multiple aggregates. A typical example is a workflow that involves several microservices. We'll see an example of this in the Drone Delivery application.
4. Finally, consider non-functional requirements. Look at factors such as team size, data types, technologies, scalability requirements, availability requirements, and security requirements. These factors may lead you to further decompose a microservice into two or more smaller services, or do the opposite and combine several microservices into one.

After you identify the microservices in your application, validate your design against the following criteria:

- Each service has a single responsibility.
- There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it may be a symptom that these functions belong in the same service.
- Each service is small enough that it can be built by a small team working independently.
- There are no inter-dependencies that will require two or more services to be deployed in lock-step. It should always be possible to deploy a service without redeploying any other services.
- Services are not tightly coupled, and can evolve independently.
- Your service boundaries will not create problems with data consistency or integrity. Sometimes it's important to maintain data consistency by putting functionality into a single microservice. That said, consider whether you really need strong

consistency. There are strategies for addressing eventual consistency in a distributed system, and the benefits of decomposing services often outweigh the challenges of managing eventual consistency.

Above all, it's important to be pragmatic, and remember that domain-driven design is an iterative process. When in doubt, start with more coarse-grained microservices. Splitting a microservice into two smaller services is easier than refactoring functionality across several existing microservices.

## Example: Defining microservices for the Drone Delivery application

Recall that the development team had identified the four aggregates — Delivery, Package, Drone, and Account — and two domain services, Scheduler and Supervisor.

Delivery and Package are obvious candidates for microservices. The Scheduler and Supervisor coordinate the activities performed by other microservices, so it makes sense to implement these domain services as microservices.

Drone and Account are interesting because they belong to other bounded contexts. One option is for the Scheduler to call the Drone and Account bounded contexts directly. Another option is to create Drone and Account microservices inside the Shipping bounded context. These microservices would mediate between the bounded contexts, by exposing APIs or data schemas that are more suited to the Shipping context.

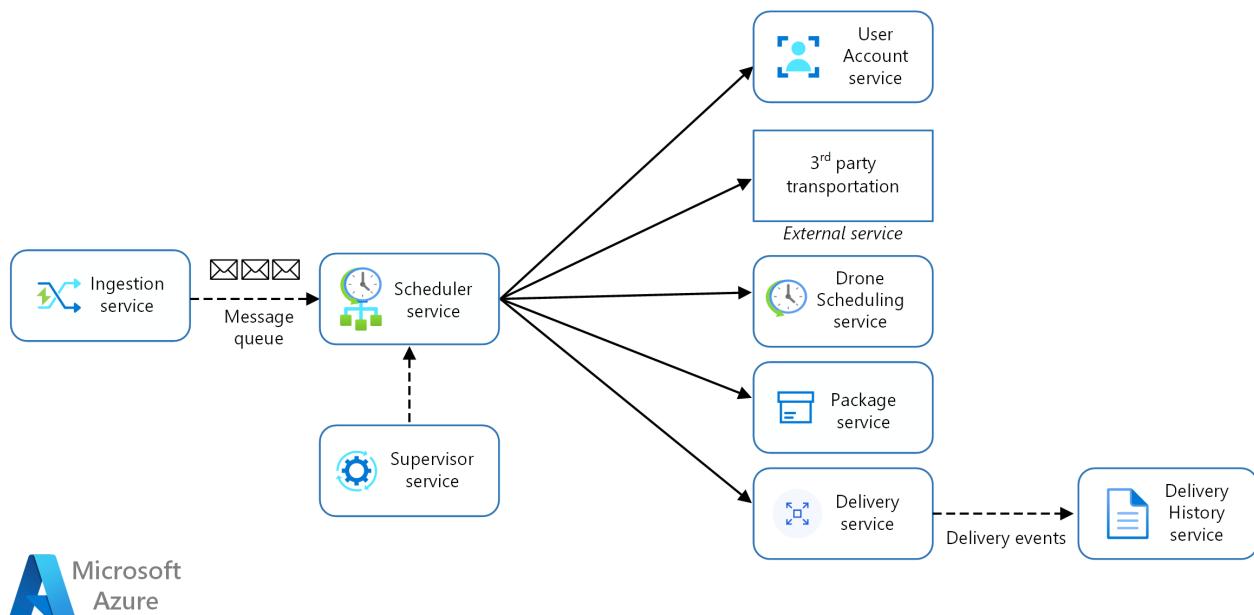
The details of the Drone and Account bounded contexts are beyond the scope of this guidance, so we created mock services for them in our reference implementation. But here are some factors to consider in this situation:

- What is the network overhead of calling directly into the other bounded context?
- Is the data schema for the other bounded context suitable for this context, or is it better to have a schema that's tailored to this bounded context?
- Is the other bounded context a legacy system? If so, you might create a service that acts as an [anti-corruption layer](#) to translate between the legacy system and the modern application.
- What is the team structure? Is it easy to communicate with the team that's responsible for the other bounded context? If not, creating a service that mediates between the two contexts can help to mitigate the cost of cross-team communication.

So far, we haven't considered any non-functional requirements. Thinking about the application's throughput requirements, the development team decided to create a separate Ingestion microservice that is responsible for ingesting client requests. This microservice will implement [load leveling](#) by putting incoming requests into a buffer for processing. The Scheduler will read the requests from the buffer and execute the workflow.

Non-functional requirements led the team to create one additional service. All of the services so far have been about the process of scheduling and delivering packages in real time. But the system also needs to store the history of every delivery in long-term storage for data analysis. The team considered making this the responsibility of the Delivery service. However, the data storage requirements are quite different for historical analysis versus in-flight operations (see [Data considerations](#)). Therefore, the team decided to create a separate Delivery History service, which will listen for DeliveryTracking events from the Delivery service and write the events into long-term storage.

The following diagram shows the design at this point:



[Download a Visio file](#) of this architecture.

## Next steps

At this point, you should have a clear understanding of the purpose and functionality of each microservice in your design. Now you can architect the system.

[Design a microservices architecture](#)

## Related resources

- [Microservices architecture design](#)
- [Using tactical DDD to design microservices](#)
- [Using domain analysis to model microservices](#)
- [Choose an Azure compute option for microservices](#)

# Design a microservices architecture

Azure Kubernetes Service (AKS)

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

In this set of articles, we explore how to build a microservices architecture on Azure. Topics include:

- [Compute options for microservices](#)
- [Interservice communication](#)
- [API design](#)
- [API gateways](#)
- [Data considerations](#)
- [Design patterns](#)

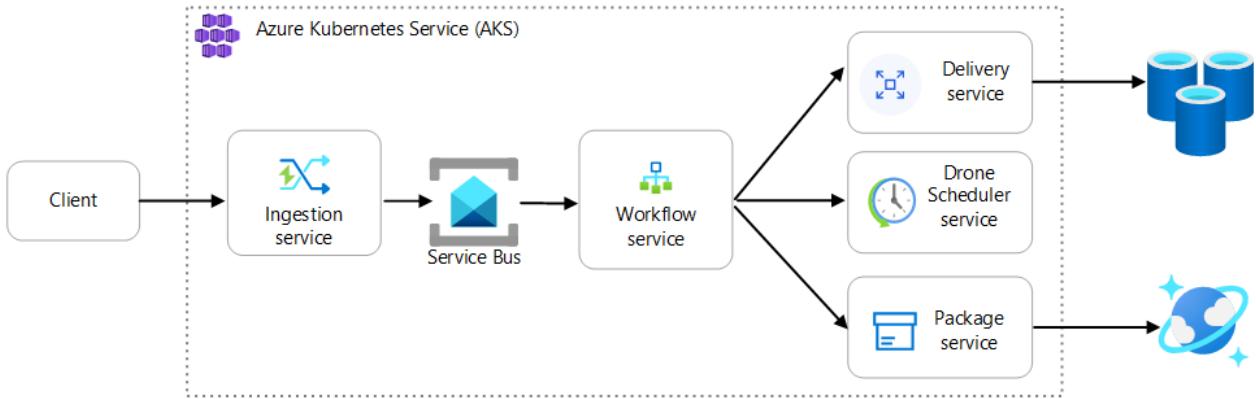
## Prerequisites

Before reading these articles, you might start with the following:

- [Introduction to microservices architectures](#). Understand the benefits and challenges of microservices, and when to use this style of architecture.
- [Using domain analysis to model microservices](#). Learn a domain-driven approach to modeling microservices.

## Reference implementation

To illustrate best practices for a microservices architecture, we created a reference implementation that we call the Drone Delivery application. This implementation runs on Kubernetes using Azure Kubernetes Service (AKS). You can find the reference implementation on [GitHub](#) .



Download a [Visio file](#) of this architecture.

## Scenario

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This solution is ideal for the aerospace and aircraft industries.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

### ⓘ Note

For help in choosing between a microservices architecture and other architectural styles, see the [Azure Application Architecture Guide](#).

Our reference implementation uses Kubernetes with [Azure Kubernetes Service \(AKS\)](#). However, many of the high-level architectural decisions and challenges will apply to any container orchestrator, including [Azure Service Fabric](#).

## Next steps

[Choose a compute option](#)

## Related resources

- [Design interservice communication for microservices](#)
- [Design APIs for microservices](#)
- [Use API gateways in microservices](#)
- [Data considerations for microservices](#)
- [Design patterns for microservices](#)

# Choose an Azure compute option for microservices

Article • 06/30/2023

The term *compute* refers to the hosting model for the computing resources that your application runs on. For a microservices architecture, two approaches are especially popular:

- A service orchestrator that manages services running on dedicated nodes (VMs).
- A serverless architecture using functions as a service (FaaS).

While these aren't the only options, they are both proven approaches to building microservices. An application might include both approaches.

## Service orchestrators

An orchestrator handles tasks related to deploying and managing a set of services. These tasks include placing services on nodes, monitoring the health of services, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling the number of instances of a service, and applying configuration updates. Popular orchestrators include Kubernetes, Service Fabric, DC/OS, and Docker Swarm.

On the Azure platform, consider the following options:

- [Azure Kubernetes Service](#) (AKS) is a managed Kubernetes service. AKS provisions Kubernetes and exposes the Kubernetes API endpoints, but hosts and manages the Kubernetes control plane, performing automated upgrades, automated patching, autoscaling, and other management tasks. You can think of AKS as being "Kubernetes APIs as a service."
- [Azure Container Apps](#) is a managed service built on Kubernetes that abstracts the complexities of container orchestration and other management tasks. Container Apps simplifies the deployment and management of containerized applications and microservices in a serverless environment while providing the features of Kubernetes.
- [Service Fabric](#) is a distributed systems platform for packaging, deploying, and managing microservices. Microservices can be deployed to Service Fabric as containers, as binary executables, or as [Reliable Services](#). Using the Reliable Services programming model, services can directly use Service Fabric programming

APIs to query the system, report health, receive notifications about configuration and code changes, and discover other services. A key differentiation with Service Fabric is its strong focus on building stateful services using [Reliable Collections](#).

- Other options such as Docker Enterprise Edition can run in an IaaS environment on Azure. You can find deployment templates on [Azure Marketplace](#) ↗.

## Containers

Sometimes people talk about containers and microservices as if they were the same thing. While that's not true — you don't need containers to build microservices — containers do have some benefits that are particularly relevant to microservices, such as:

- **Portability.** A container image is a standalone package that runs without needing to install libraries or other dependencies. That makes them easy to deploy. Containers can be started and stopped quickly, so you can spin up new instances to handle more load or to recover from node failures.
- **Density.** Containers are lightweight compared with running a virtual machine, because they share OS resources. That makes it possible to pack multiple containers onto a single node, which is especially useful when the application consists of many small services.
- **Resource isolation.** You can limit the amount of memory and CPU that is available to a container, which can help to ensure that a runaway process doesn't exhaust the host resources. See the [Bulkhead pattern](#) for more information.

## Serverless (Functions as a Service)

With a [serverless](#) ↗ architecture, you don't manage the VMs or the virtual network infrastructure. Instead, you deploy code and the hosting service handles putting that code onto a VM and executing it. This approach tends to favor small granular functions that are coordinated using event-based triggers. For example, a message being placed onto a queue might trigger a function that reads from the queue and processes the message.

[Azure Functions](#) is a serverless compute service that supports various function triggers, including HTTP requests, Service Bus queues, and Event Hubs events. For a complete list, see [Azure Functions triggers and bindings concepts](#). Also consider [Azure Event Grid](#), which is a managed event routing service in Azure.

# Orchestrator or serverless?

Here are some factors to consider when choosing between an orchestrator approach and a serverless approach.

**Manageability** A serverless application is easy to manage, because the platform manages all compute resources for you. While an orchestrator abstracts some aspects of managing and configuring a cluster, it does not completely hide the underlying VMs. With an orchestrator, you will need to think about issues such as load balancing, CPU and memory usage, and networking.

**Flexibility and control.** An orchestrator gives you a great deal of control over configuring and managing your services and the cluster. The tradeoff is additional complexity. With a serverless architecture, you give up some degree of control because these details are abstracted.

**Portability.** All of the orchestrators listed here (Kubernetes, DC/OS, Docker Swarm, and Service Fabric) can run on-premises or in multiple public clouds.

**Application integration.** It can be challenging to build a complex application using a serverless architecture, due to the need to coordinate, deploy, and manage many small independent functions. One option in Azure is to use [Azure Logic Apps](#) to coordinate a set of Azure Functions. For an example of this approach, see [Create a function that integrates with Azure Logic Apps](#).

**Cost.** With an orchestrator, you pay for the VMs that are running in the cluster. With a serverless application, you pay only for the actual compute resources consumed. In both cases, you need to factor in the cost of any additional services, such as storage, databases, and messaging services.

**Scalability.** Azure Functions scales automatically to meet demand, based on the number of incoming events. With an orchestrator, you can scale out by increasing the number of service instances running in the cluster. You can also scale by adding additional VMs to the cluster.

Our reference implementation primarily uses Kubernetes, but we did use Azure Functions for one service, namely the Delivery History service. Azure Functions was a good fit for this particular service, because it's an event-driven workload. By using an Event Hubs trigger to invoke the function, the service needed a minimal amount of code. Also, the Delivery History service is not part of the main workflow, so running it outside of the Kubernetes cluster doesn't affect the end-to-end latency of user-initiated operations.

# Next steps

Interservice communication

## Related resources

- Using domain analysis to model microservices
- Design a microservices architecture
- Expose Azure Spring Apps through a reverse proxy
- Design APIs for microservices

# Design interservice communication for microservices

Azure DevOps

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single business activity, this can be a challenge. In this article, we look at the tradeoffs between asynchronous messaging versus synchronous APIs. Then we look at some of the challenges in designing resilient interservice communication.

## Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this article, are designed to handle many of these challenges.

**Resiliency.** There may be dozens or even hundreds of instances of any given microservice. An instance can fail for any number of reasons. There can be a node-level failure, such as a hardware failure or a VM reboot. An instance might crash, or be overwhelmed with requests and unable to process any new requests. Any of these events can cause a network call to fail. There are two design patterns that can help make service-to-service network calls more resilient:

- **Retry.** A network call may fail because of a transient fault that goes away by itself. Rather than fail outright, the caller should typically retry the operation a certain number of times, or until a configured time-out period elapses. However, if an operation is not idempotent, retries can cause unintended side effects. The original call might succeed, but the caller never gets a response. If the caller retries, the operation may be invoked twice. Generally, it's not safe to retry POST or PATCH methods, because these are not guaranteed to be idempotent.
- **Circuit Breaker.** Too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. The Circuit Breaker pattern can prevent a service from repeatedly trying an operation that is likely to fail.

**Load balancing.** When service "A" calls service "B", the request must reach a running instance of service "B". In Kubernetes, the `Service` resource type provides a stable IP address for a group of pods. Network traffic to the service's IP address gets forwarded to a pod by means of iptable rules. By default, a random pod is chosen. A service mesh (see below) can provide more intelligent load balancing algorithms based on observed latency or other metrics.

**Distributed tracing.** A single transaction may span multiple services. That can make it hard to monitor the overall performance and health of the system. Even if every service generates logs and metrics, without some way to tie them together, they are of limited use. The article [Logging and monitoring](#) talks more about distributed tracing, but we mention it here as a challenge.

**Service versioning.** When a team deploys a new version of a service, they must avoid breaking any other services or external clients that depend on it. In addition, you might want to run multiple versions of a service side-by-side, and route requests to a particular version. See [API Versioning](#) for more discussion of this issue.

**TLS encryption and mutual TLS authentication.** For security reasons, you may want to encrypt traffic between services with TLS, and use mutual TLS authentication to authenticate callers.

## Synchronous versus asynchronous messaging

There are two basic messaging patterns that microservices can use to communicate with other microservices.

1. Synchronous communication. In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP or gRPC. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.
2. Asynchronous message passing. In this pattern, a service sends message without waiting for a response, and one or more services process the message asynchronously.

It's important to distinguish between asynchronous I/O and an asynchronous protocol. Asynchronous I/O means the calling thread is not blocked while the I/O completes. That's important for performance, but is an implementation detail in terms of the architecture. An asynchronous protocol means the sender doesn't wait for a response. HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

There are tradeoffs to each pattern. Request/response is a well-understood paradigm, so designing an API may feel more natural than designing a messaging system. However, asynchronous messaging has some advantages that can be useful in a microservices architecture:

- **Reduced coupling.** The message sender does not need to know about the consumer.
- **Multiple subscribers.** Using a pub/sub model, multiple consumers can subscribe to receive events. See [Event-driven architecture style](#).
- **Failure isolation.** If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in a microservices architecture, because each service has its own lifecycle. A service could become unavailable or be replaced with a newer version at any given time. Asynchronous messaging can handle intermittent downtime. Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.
- **Responsiveness.** An upstream service can reply faster if it does not wait on downstream services. This is especially useful in a microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.
- **Load leveling.** A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.
- **Workflows.** Queues can be used to manage a workflow, by check-pointing the message after each step in the workflow.

However, there are also some challenges to using asynchronous messaging effectively.

- **Coupling with the messaging infrastructure.** Using a particular messaging infrastructure may cause tight coupling with that infrastructure. It will be difficult to switch to another messaging infrastructure later.
- **Latency.** End-to-end latency for an operation may become high if the message queues fill up.
- **Cost.** At high throughputs, the monetary cost of the messaging infrastructure could be significant.
- **Complexity.** Handling asynchronous messaging is not a trivial task. For example, you must handle duplicated messages, either by de-duplicating or by making operations idempotent. It's also hard to implement request-response semantics

using asynchronous messaging. To send a response, you need another queue, plus a way to correlate request and response messages.

- **Throughput.** If messages require *queue semantics*, the queue can become a bottleneck in the system. Each message requires at least one queue operation and one dequeue operation. Moreover, queue semantics generally require some kind of locking inside the messaging infrastructure. If the queue is a managed service, there may be additional latency, because the queue is external to the cluster's virtual network. You can mitigate these issues by batching messages, but that complicates the code. If the messages don't require queue semantics, you might be able to use an event *stream* instead of a queue. For more information, see [Event-driven architectural style](#).

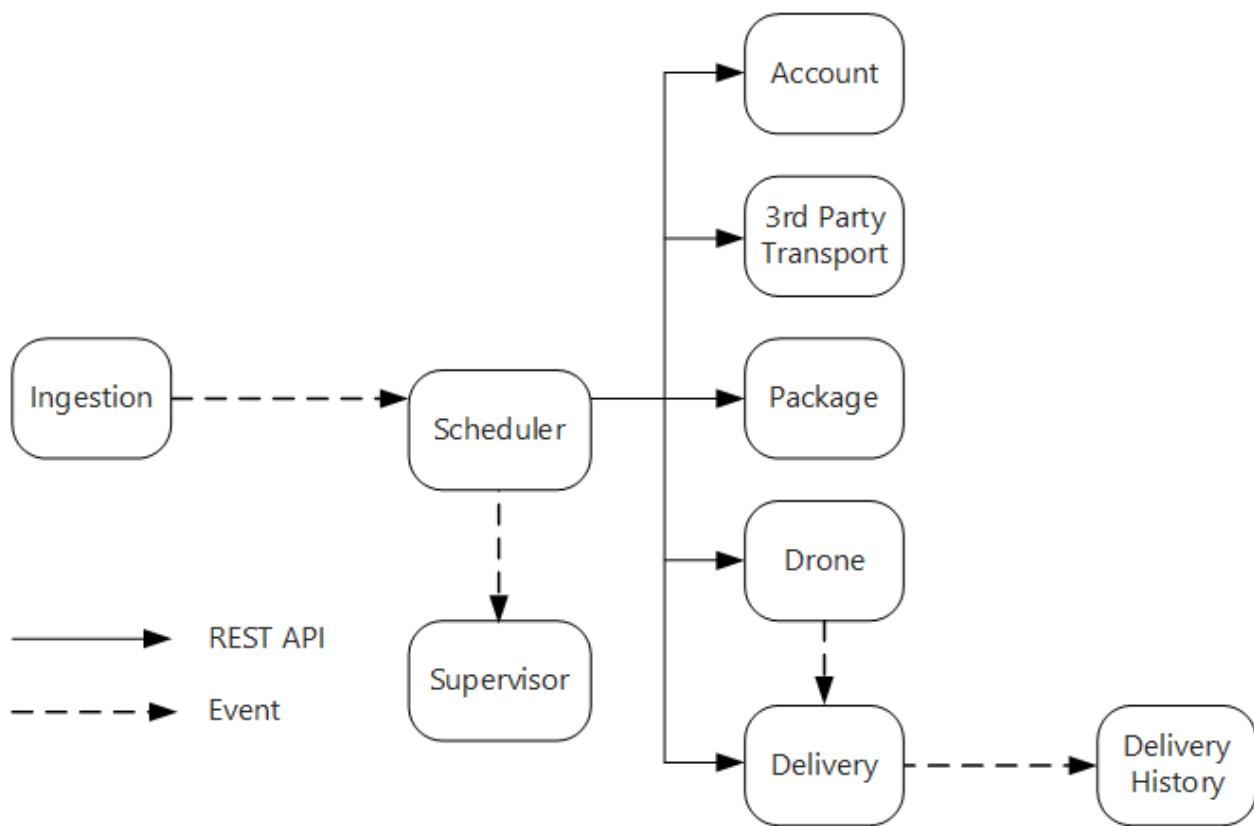
## Drone Delivery: Choosing the messaging patterns

This solution uses the Drone Delivery example. It's ideal for the aerospace and aircraft industries.

With these considerations in mind, the development team made the following design choices for the Drone Delivery application:

- The Ingestion service exposes a public REST API that client applications use to schedule, update, or cancel deliveries.
- The Ingestion service uses Event Hubs to send asynchronous messages to the Scheduler service. Asynchronous messages are necessary to implement the load-leveling that is required for ingestion.
- The Account, Delivery, Package, Drone, and Third-party Transport services all expose internal REST APIs. The Scheduler service calls these APIs to carry out a user request. One reason to use synchronous APIs is that the Scheduler needs to get a response from each of the downstream services. A failure in any of the downstream services means the entire operation failed. However, a potential issue is the amount of latency that is introduced by calling the backend services.
- If any downstream service has a nontransient failure, the entire transaction should be marked as failed. To handle this case, the Scheduler service sends an asynchronous message to the Supervisor, so that the Supervisor can schedule compensating transactions.

- The Delivery service exposes a public API that clients can use to get the status of a delivery. In the article [API gateway](#), we discuss how an API gateway can hide the underlying services from the client, so the client doesn't need to know which services expose which APIs.
- While a drone is in flight, the Drone service sends events that contain the drone's current location and status. The Delivery service listens to these events in order to track the status of a delivery.
- When the status of a delivery changes, the Delivery service sends a delivery status event, such as `DeliveryCreated` or `DeliveryCompleted`. Any service can subscribe to these events. In the current design, the Delivery History service is the only subscriber, but there might be other subscribers later. For example, the events might go to a real-time analytics service. And because the Scheduler doesn't have to wait for a response, adding more subscribers doesn't affect the main workflow path.



Notice that delivery status events are derived from drone location events. For example, when a drone reaches a delivery location and drops off a package, the Delivery service translates this into a `DeliveryCompleted` event. This is an example of thinking in terms of domain models. As described earlier, Drone Management belongs in a separate bounded context. The drone events convey the physical location of a drone. The delivery events, on the other hand, represent changes in the status of a delivery, which is a different business entity.

# Using a service mesh

A *service mesh* is a software layer that handles service-to-service communication. Service meshes are designed to address many of the concerns listed in the previous section, and to move responsibility for these concerns away from the microservices themselves and into a shared layer. The service mesh acts as a proxy that intercepts network communication between microservices in the cluster. Currently, the service mesh concept applies mainly to container orchestrators, rather than serverless architectures.

## ⓘ Note

Service mesh is an example of the **Ambassador pattern** — a helper service that sends network requests on behalf of the application.

Right now, the main options for a service mesh in Kubernetes are [Linkerd](#) and [Istio](#). Both of these technologies are evolving rapidly. However, some features that both Linkerd and Istio have in common include:

- Load balancing at the session level, based on observed latencies or number of outstanding requests. This can improve performance over the layer-4 load balancing that is provided by Kubernetes.
- Layer-7 routing based on URL path, Host header, API version, or other application-level rules.
- Retry of failed requests. A service mesh understands HTTP error codes, and can automatically retry failed requests. You can configure that maximum number of retries, along with a timeout period in order to bound the maximum latency.
- Circuit breaking. If an instance consistently fails requests, the service mesh will temporarily mark it as unavailable. After a backoff period, it will try the instance again. You can configure the circuit breaker based on various criteria, such as the number of consecutive failures,
- Service mesh captures metrics about interservice calls, such as the request volume, latency, error and success rates, and response sizes. The service mesh also enables distributed tracing by adding correlation information for each hop in a request.
- Mutual TLS Authentication for service-to-service calls.

Do you need a service mesh? It depends. Without a service mesh, you'll need to consider each of the challenges mentioned at the beginning of this article. You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a

service mesh moves these concerns out of the individual services and into a dedicated layer. On the other hand, a service mesh adds complexity to the setup and configuration of the cluster. There may be performance implications, because requests now get routed through the service mesh proxy, and because extra services are now running on every node in the cluster. You should do thorough performance and load testing before deploying a service mesh in production.

## Distributed transactions

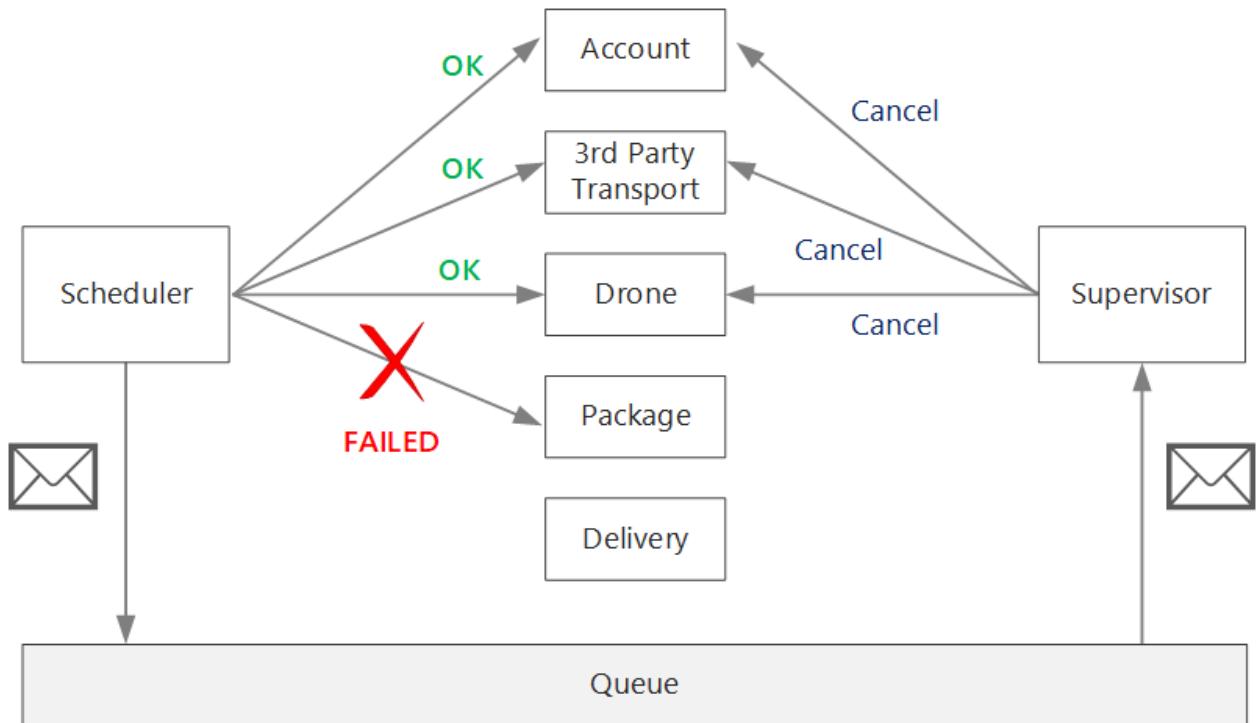
A common challenge in microservices is correctly handling transactions that span multiple services. Often in this scenario, the success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

There are two cases to consider:

- A service may experience a *transient* failure such as a network timeout. These errors can often be resolved simply by retrying the call. If the operation still fails after a certain number of attempts, it's considered a nontransient failure.
- A *nontransient* failure is any failure that's unlikely to go away by itself. Nontransient failures include normal error conditions, such as invalid input. They also include unhandled exceptions in application code or a process crashing. If this type of error occurs, the entire business transaction must be marked as a failure. It may be necessary to undo other steps in the same transaction that already succeeded.

After a nontransient failure, the current transaction might be in a *partially failed* state, where one or more steps already completed successfully. For example, if the Drone service already scheduled a drone, the drone must be canceled. In that case, the application needs to undo the steps that succeeded, by using a [Compensating Transaction](#). In some cases, this must be done by an external system or even by a manual process.

If the logic for compensating transactions is complex, consider creating a separate service that is responsible for this process. In the Drone Delivery application, the Scheduler service puts failed operations onto a dedicated queue. A separate microservice, called the Supervisor, reads from this queue and calls a cancellation API on the services that need to compensate. This is a variation of the [Scheduler Agent Supervisor pattern](#). The Supervisor service might take other actions as well, such as notify the user by text or email, or send an alert to an operations dashboard.



The Scheduler service itself might fail (for example, because a node crashes). In that case, a new instance can spin up and take over. However, any transactions that were already in progress must be resumed.

One approach is to save a checkpoint to a durable store after each step in the workflow is completed. If an instance of the Scheduler service crashes in the middle of a transaction, a new instance can use the checkpoint to resume where the previous instance left off. However, writing checkpoints can create a performance overhead.

Another option is to design all operations to be idempotent. An operation is idempotent if it can be called multiple times without producing additional side-effects after the first call. Essentially, the downstream service should ignore duplicate calls, which means the service must be able to detect duplicate calls. It's not always straightforward to implement idempotent methods. For more information, see [Idempotent operations](#).

## Next steps

For microservices that talk directly to each other, it's important to create well-designed APIs.

[API design](#)

## Related resources

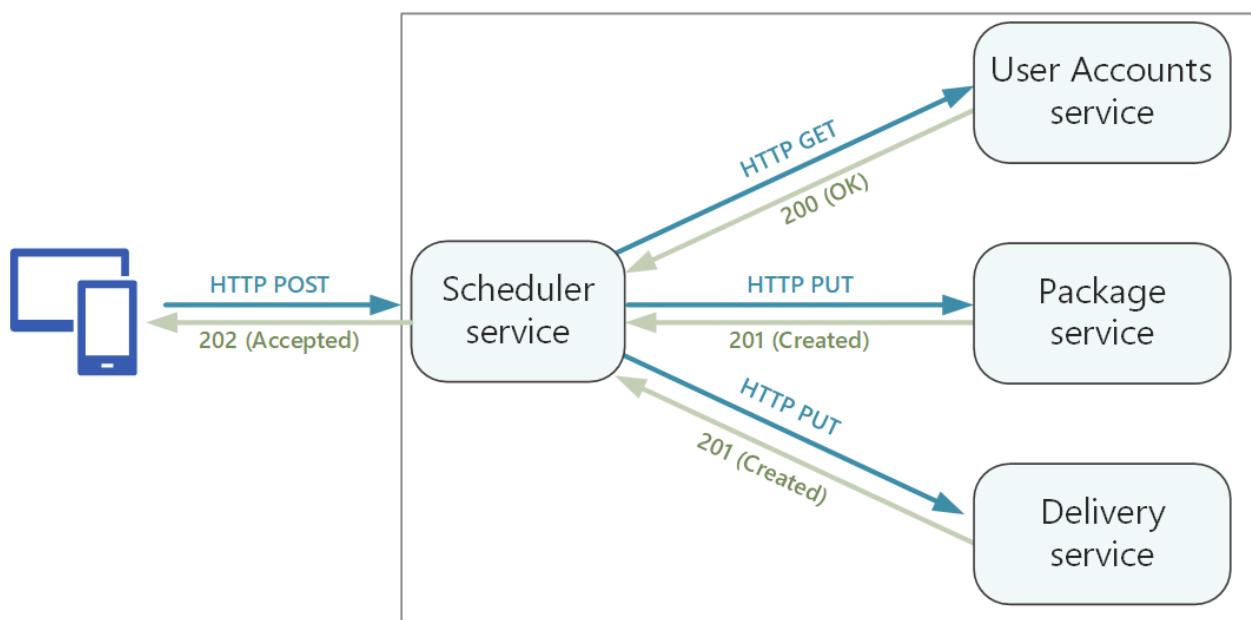
- [Design a microservices architecture](#)
- [Using domain analysis to model microservices](#)

- Using tactical DDD to design microservices
- Identify microservice boundaries

# Design APIs for microservices

Azure DevOps

Good API design is important in a microservices architecture, because all data exchange between services happens either through messages or API calls. APIs must be efficient to avoid creating [chatty I/O](#). Because services are designed by teams working independently, APIs must have well-defined semantics and versioning schemes, so that updates don't break other services.



It's important to distinguish between two types of API:

- Public APIs that client applications call.
- Backend APIs that are used for interservice communication.

These two use cases have somewhat different requirements. A public API must be compatible with client applications, typically browser applications or native mobile applications. Most of the time, that means the public API will use REST over HTTP. For the backend APIs, however, you need to take network performance into account.

Depending on the granularity of your services, interservice communication can result in a lot of network traffic. Services can quickly become I/O bound. For that reason, considerations such as serialization speed and payload size become more important. Some popular alternatives to using REST over HTTP include gRPC, Apache Avro, and Apache Thrift. These protocols support binary serialization and are generally more efficient than HTTP.

## Considerations

Here are some things to think about when choosing how to implement an API.

**REST versus RPC.** Consider the tradeoffs between using a REST-style interface versus an RPC-style interface.

- REST models resources, which can be a natural way to express your domain model. It defines a uniform interface based on HTTP verbs, which encourages evolvability. It has well-defined semantics in terms of idempotency, side effects, and response codes. And it enforces stateless communication, which improves scalability.
- RPC is more oriented around operations or commands. Because RPC interfaces look like local method calls, it may lead you to design overly chatty APIs. However, that doesn't mean RPC must be chatty. It just means you need to use care when designing the interface.

For a RESTful interface, the most common choice is REST over HTTP using JSON. For an RPC-style interface, there are several popular frameworks, including gRPC, Apache Avro, and Apache Thrift.

**Efficiency.** Consider efficiency in terms of speed, memory, and payload size. Typically a gRPC-based interface is faster than REST over HTTP.

**Interface definition language (IDL).** An IDL is used to define the methods, parameters, and return values of an API. An IDL can be used to generate client code, serialization code, and API documentation. IDLs can also be consumed by API testing tools such as Postman. Frameworks such as gRPC, Avro, and Thrift define their own IDL specifications. REST over HTTP does not have a standard IDL format, but a common choice is OpenAPI (formerly Swagger). You can also create an HTTP REST API without using a formal definition language, but then you lose the benefits of code generation and testing.

**Serialization.** How are objects serialized over the wire? Options include text-based formats (primarily JSON) and binary formats such as protocol buffer. Binary formats are generally faster than text-based formats. However, JSON has advantages in terms of interoperability, because most languages and frameworks support JSON serialization. Some serialization formats require a fixed schema, and some require compiling a schema definition file. In that case, you'll need to incorporate this step into your build process.

**Framework and language support.** HTTP is supported in nearly every framework and language. gRPC, Avro, and Thrift all have libraries for C++, C#, Java, and Python. Thrift and gRPC also support Go.

**Compatibility and interoperability.** If you choose a protocol like gRPC, you may need a protocol translation layer between the public API and the back end. A [gateway](#) can

perform that function. If you are using a service mesh, consider which protocols are compatible with the service mesh. For example, Linkerd has built-in support for HTTP, Thrift, and gRPC.

Our baseline recommendation is to choose REST over HTTP unless you need the performance benefits of a binary protocol. REST over HTTP requires no special libraries. It creates minimal coupling, because callers don't need a client stub to communicate with the service. There are rich ecosystems of tools to support schema definitions, testing, and monitoring of RESTful HTTP endpoints. Finally, HTTP is compatible with browser clients, so you don't need a protocol translation layer between the client and the backend.

However, if you choose REST over HTTP, you should do performance and load testing early in the development process, to validate whether it performs well enough for your scenario.

## RESTful API design

There are many resources for designing RESTful APIs. Here are some that you might find helpful:

- [API design](#)
- [API implementation](#)
- [Microsoft REST API Guidelines](#) ↗

Here are some specific considerations to keep in mind.

- Watch out for APIs that leak internal implementation details or simply mirror an internal database schema. The API should model the domain. It's a contract between services, and ideally should only change when new functionality is added, not just because you refactored some code or normalized a database table.
- Different types of client, such as mobile application and desktop web browser, may require different payload sizes or interaction patterns. Consider using the [Backends for Frontends pattern](#) to create separate backends for each client, which expose an optimal interface for that client.
- For operations with side effects, consider making them idempotent and implementing them as PUT methods. That will enable safe retries and can improve resiliency. The article [Interservice communication](#) discuss this issue in more detail.

- HTTP methods can have asynchronous semantics, where the method returns a response immediately, but the service carries out the operation asynchronously. In that case, the method should return an [HTTP 202 ↗](#) response code, which indicates the request was accepted for processing, but the processing is not yet completed. For more information, see [Asynchronous Request-Reply pattern](#).

## Mapping REST to DDD patterns

Patterns such as entity, aggregate, and value object are designed to place certain constraints on the objects in your domain model. In many discussions of DDD, the patterns are modeled using object-oriented (OO) language concepts like constructors or property getters and setters. For example, *value objects* are supposed to be immutable. In an OO programming language, you would enforce this by assigning the values in the constructor and making the properties read-only:

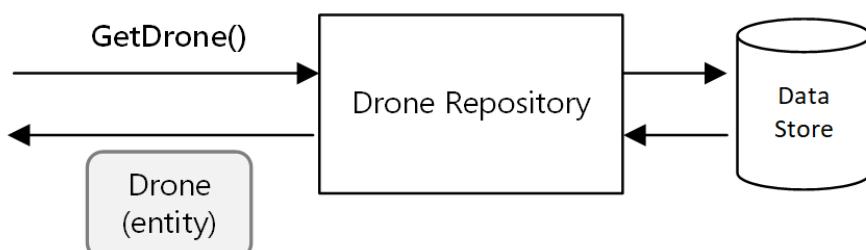
ts

```
export class Location {
 readonly latitude: number;
 readonly longitude: number;

 constructor(latitude: number, longitude: number) {
 if (latitude < -90 || latitude > 90) {
 throw new RangeError('latitude must be between -90 and 90');
 }
 if (longitude < -180 || longitude > 180) {
 throw new RangeError('longitude must be between -180 and 180');
 }
 this.latitude = latitude;
 this.longitude = longitude;
 }
}
```

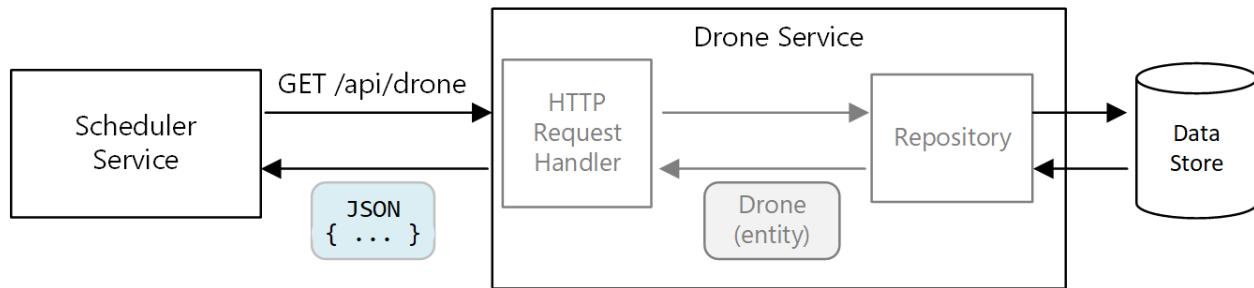
These sorts of coding practices are particularly important when building a traditional monolithic application. With a large code base, many subsystems might use the `Location` object, so it's important for the object to enforce correct behavior.

Another example is the Repository pattern, which ensures that other parts of the application do not make direct reads or writes to the data store:



In a microservices architecture, however, services don't share the same code base and don't share data stores. Instead, they communicate through APIs. Consider the case where the Scheduler service requests information about a drone from the Drone service. The Drone service has its internal model of a drone, expressed through code. But the Scheduler doesn't see that. Instead, it gets back a *representation* of the drone entity — perhaps a JSON object in an HTTP response.

This example is ideal for the aircraft and aerospace industries.



The Scheduler service can't modify the Drone service's internal models, or write to the Drone service's data store. That means the code that implements the Drone service has a smaller exposed surface area, compared with code in a traditional monolith. If the Drone service defines a Location class, the scope of that class is limited — no other service will directly consume the class.

For these reasons, this guidance doesn't focus much on coding practices as they relate to the tactical DDD patterns. But it turns out that you can also model many of the DDD patterns through REST APIs.

For example:

- Aggregates map naturally to *resources* in REST. For example, the Delivery aggregate would be exposed as a resource by the Delivery API.
- Aggregates are consistency boundaries. Operations on aggregates should never leave an aggregate in an inconsistent state. Therefore, you should avoid creating APIs that allow a client to manipulate the internal state of an aggregate. Instead, favor coarse-grained APIs that expose aggregates as resources.
- Entities have unique identities. In REST, resources have unique identifiers in the form of URLs. Create resource URLs that correspond to an entity's domain identity. The mapping from URL to domain identity may be opaque to client.
- Child entities of an aggregate can be reached by navigating from the root entity. If you follow [HATEOAS](#) principles, child entities can be reached via links in the representation of the parent entity.

- Because value objects are immutable, updates are performed by replacing the entire value object. In REST, implement updates through PUT or PATCH requests.
- A repository lets clients query, add, or remove objects in a collection, abstracting the details of the underlying data store. In REST, a collection can be a distinct resource, with methods for querying the collection or adding new entities to the collection.

When you design your APIs, think about how they express the domain model, not just the data inside the model, but also the business operations and the constraints on the data.

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

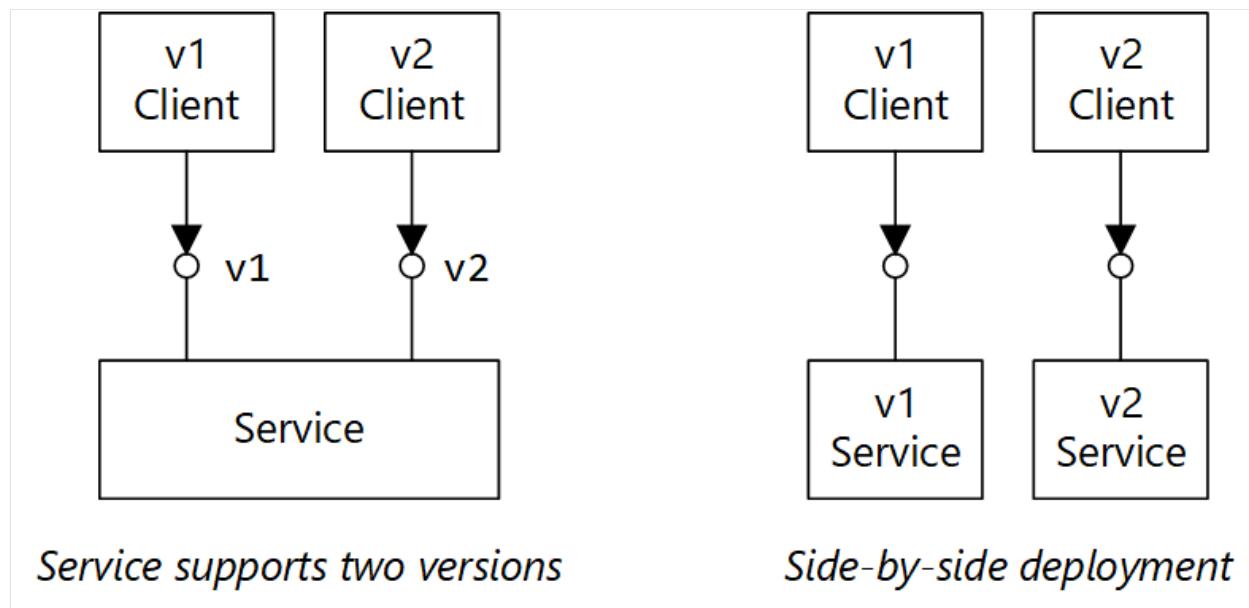
DDD concept	REST equivalent	Example
Aggregate	Resource	<code>{ "1":1234, "status":"pending" ... }</code>
Identity	URL	<code>https://delivery-service/deliveries/1</code>
Child entities	Links	<code>{ "href": "/deliveries/1/confirmation" }</code>
Update value objects	PUT or PATCH	<code>PUT https://delivery-service/deliveries/1/dropoff</code>
Repository	Collection	<code>https://delivery-service/deliveries?status=pending</code>

## API versioning

An API is a contract between a service and clients or consumers of that service. If an API changes, there is a risk of breaking clients that depend on the API, whether those are external clients or other microservices. Therefore, it's a good idea to minimize the number of API changes that you make. Often, changes in the underlying implementation don't require any changes to the API. Realistically, however, at some point you will want to add new features or new capabilities that require changing an existing API.

Whenever possible, make API changes backward compatible. For example, avoid removing a field from a model, because that can break clients that expect the field to be there. Adding a field does not break compatibility, because clients should ignore any fields they don't understand in a response. However, the service must handle the case where an older client omits the new field in a request.

Support versioning in your API contract. If you introduce a breaking API change, introduce a new API version. Continue to support the previous version, and let clients select which version to call. There are a couple of ways to do this. One is simply to expose both versions in the same service. Another option is to run two versions of the service side-by-side, and route requests to one or the other version, based on HTTP routing rules.



There's a cost to supporting multiple versions, in terms of developer time, testing, and operational overhead. Therefore, it's good to deprecate old versions as quickly as possible. For internal APIs, the team that owns the API can work with other teams to help them migrate to the new version. This is when having a cross-team governance process is useful. For external (public) APIs, it can be harder to deprecate an API version, especially if the API is consumed by third parties or by native client applications.

When a service implementation changes, it's useful to tag the change with a version. The version provides important information when troubleshooting errors. It can be very helpful for root cause analysis to know exactly which version of the service was called. Consider using [semantic versioning](#) for service versions. Semantic versioning uses a `MAJOR.MINOR.PATCH` format. However, clients should only select an API by the major version number, or possibly the minor version if there are significant (but non-breaking) changes between minor versions. In other words, it's reasonable for clients to select between version 1 and version 2 of an API, but not to select version 2.1.3. If you allow that level of granularity, you risk having to support a proliferation of versions.

For further discussion of API versioning, see [Versioning a RESTful web API](#).

## Idempotent operations

An operation is *idempotent* if it can be called multiple times without producing additional side-effects after the first call. Idempotency can be a useful resiliency strategy, because it allows an upstream service to safely invoke an operation multiple times. For a discussion of this point, see [Distributed transactions](#).

The HTTP specification states that GET, PUT, and DELETE methods must be idempotent. POST methods are not guaranteed to be idempotent. If a POST method creates a new resource, there is generally no guarantee that this operation is idempotent. The specification defines idempotent this way:

A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. ([RFC 7231](#) ↗)

It's important to understand the difference between PUT and POST semantics when creating a new entity. In both cases, the client sends a representation of an entity in the request body. But the meaning of the URI is different.

- For a POST method, the URI represents a parent resource of the new entity, such as a collection. For example, to create a new delivery, the URI might be `/api/deliveries`. The server creates the entity and assigns it a new URI, such as `/api/deliveries/39660`. This URI is returned in the Location header of the response. Each time the client sends a request, the server will create a new entity with a new URI.
- For a PUT method, the URI identifies the entity. If there already exists an entity with that URI, the server replaces the existing entity with the version in the request. If no entity exists with that URI, the server creates one. For example, suppose the client sends a PUT request to `api/deliveries/39660`. Assuming there is no delivery with that URI, the server creates a new one. Now if the client sends the same request again, the server will replace the existing entity.

Here is the Delivery service's implementation of the PUT method.

C#

```
[HttpPut("{id}")]
[ProducesResponseType(typeof(Delivery), 201)]
[ProducesResponseType(typeof(void), 204)]
```

```

public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)
{
 logger.LogInformation("In Put action with delivery {Id}:
{@DeliveryInfo}", id, delivery.ToLogInfo());
 try
 {
 var internalDelivery = delivery.ToInternal();

 // Create the new delivery entity.
 await deliveryRepository.CreateAsync(internalDelivery);

 // Create a delivery status event.
 var deliveryStatusEvent = new DeliveryStatusEvent { DeliveryId =
delivery.Id, Stage = DeliveryEventType.Created };
 await deliveryStatusEventRepository.AddAsync(deliveryStatusEvent);

 // Return HTTP 201 (Created)
 return CreatedAtRoute("GetDelivery", new { id= delivery.Id },
delivery);
 }
 catch (DuplicateResourceException)
 {
 // This method is mainly used to create deliveries. If the delivery
already exists then update it.
 logger.LogInformation("Updating resource with delivery id:
{DeliveryId}", id);

 var internalDelivery = delivery.ToInternal();
 await deliveryRepository.UpdateAsync(id, internalDelivery);

 // Return HTTP 204 (No Content)
 return NoContent();
 }
}

```

It's expected that most requests will create a new entity, so the method optimistically calls `CreateAsync` on the repository object, and then handles any duplicate-resource exceptions by updating the resource instead.

## Next steps

Learn about using an API gateway at the boundary between client applications and microservices.

[API gateways](#)

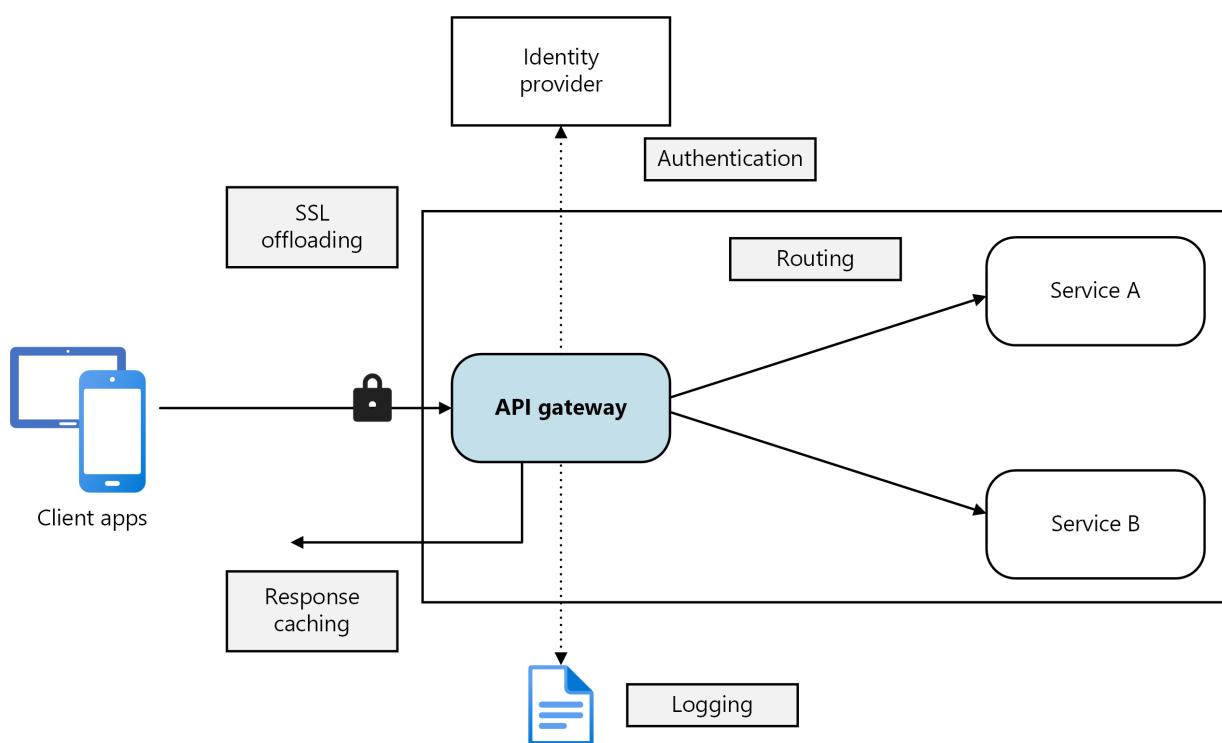
## Related resources

- RESTful web API design
- API implementation
- Design a microservices architecture
- Using domain analysis to model microservices

# Use API gateways in microservices

Azure Application Gateway    Azure API Management

In a microservices architecture, a client might interact with more than one front-end service. Given this fact, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? How do services handle SSL termination, authentication, and other concerns? An *API gateway* can help to address these challenges.



Download a [Visio file](#) of this architecture.

## What is an API gateway?

An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting. If you don't deploy a gateway, clients must send requests directly to front-end services. However, there are some potential problems with exposing services directly to clients:

- It can result in complex client code. The client must keep track of multiple endpoints, and handle failures in a resilient way.
- It creates coupling between the client and the backend. The client needs to know how the individual services are decomposed. That makes it harder to maintain the client and also harder to refactor services.
- A single operation might require calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency.
- Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting.
- Services must expose a client-friendly protocol such as HTTP or WebSocket. This limits the choice of [communication protocols](#).
- Services with public endpoints are a potential attack surface, and must be hardened.

A gateway helps to address these issues by decoupling clients from services. Gateways can perform a number of different functions, and you may not need all of them. The functions can be grouped into the following design patterns:

[Gateway Routing](#). Use the gateway as a reverse proxy to route requests to one or more backend services, using layer 7 routing. The gateway provides a single endpoint for clients, and helps to decouple clients from services.

[Gateway Aggregation](#). Use the gateway to aggregate multiple individual requests into a single request. This pattern applies when a single operation requires calls to multiple backend services. The client sends one request to the gateway. The gateway dispatches requests to the various backend services, and then aggregates the results and sends them back to the client. This helps to reduce chattiness between the client and the backend.

[Gateway Offloading](#). Use the gateway to offload functionality from individual services to the gateway, particularly cross-cutting concerns. It can be useful to consolidate these functions into one place, rather than making every service responsible for implementing them. This is particularly true for features that require specialized skills to implement correctly, such as authentication and authorization.

Here are some examples of functionality that could be offloaded to a gateway:

- SSL termination
- Authentication
- IP allowlist or blocklist
- Client rate limiting (throttling)
- Logging and monitoring

- Response caching
- Web application firewall
- GZIP compression
- Servicing static content

## Choosing a gateway technology

Here are some options for implementing an API gateway in your application.

- **Reverse proxy server.** Nginx and HAProxy are popular reverse proxy servers that support features such as load balancing, SSL, and layer 7 routing. They are both free, open-source products, with paid editions that provide additional features and support options. Nginx and HAProxy are both mature products with rich feature sets and high performance. You can extend them with third-party modules or by writing custom scripts in Lua. Nginx also supports a JavaScript-based scripting module referred to as [NGINX JavaScript](#). This module was formally named `nginxScript`.
- **Service mesh ingress controller.** If you are using a service mesh such as Linkerd or Istio, consider the features that are provided by the ingress controller for that service mesh. For example, the Istio ingress controller supports layer 7 routing, HTTP redirects, retries, and other features.
- [Azure Application Gateway](#). Application Gateway is a managed load balancing service that can perform layer-7 routing and SSL termination. It also provides a web application firewall (WAF).
- [Azure Front Door](#) is Microsoft's modern cloud Content Delivery Network (CDN) that provides fast, reliable, and secure access between your users and your applications' static and dynamic web content across the globe. Azure Front Door delivers your content using the Microsoft's global edge network with hundreds of global and local points of presence (PoPs) distributed around the world close to both your enterprise and consumer end users.
- [Azure API Management](#). API Management is a turnkey solution for publishing APIs to external and internal customers. It provides features that are useful for managing a public-facing API, including rate limiting, IP restrictions, and authentication using Microsoft Entra ID or other identity providers. API Management doesn't perform any load balancing, so it should be used in conjunction with a load balancer such as Application Gateway or a reverse proxy. For information about using API Management with Application Gateway, see [Integrate API Management in an internal VNet with Application Gateway](#).

When choosing a gateway technology, consider the following:

**Features.** The options listed above all support layer 7 routing, but support for other features will vary. Depending on the features that you need, you might deploy more than one gateway.

**Deployment.** Azure Application Gateway and API Management are managed services. Nginx and HAProxy will typically run in containers inside the cluster, but can also be deployed to dedicated VMs outside of the cluster. This isolates the gateway from the rest of the workload, but incurs higher management overhead.

**Management.** When services are updated or new services are added, the gateway routing rules might need to be updated. Consider how this process will be managed. Similar considerations apply to managing SSL certificates, IP allowlists, and other aspects of configuration.

## Deploying Nginx or HAProxy to Kubernetes

You can deploy Nginx or HAProxy to Kubernetes as a [ReplicaSet](#) or [DaemonSet](#) that specifies the Nginx or HAProxy container image. Use a ConfigMap to store the configuration file for the proxy, and mount the ConfigMap as a volume. Create a service of type LoadBalancer to expose the gateway through an Azure Load Balancer.

An alternative is to create an Ingress Controller. An Ingress Controller is a Kubernetes resource that deploys a load balancer or reverse proxy server. Several implementations exist, including Nginx and HAProxy. A separate resource called an Ingress defines settings for the Ingress Controller, such as routing rules and TLS certificates. That way, you don't need to manage complex configuration files that are specific to a particular proxy server technology.

The gateway is a potential bottleneck or single point of failure in the system, so always deploy at least two replicas for high availability. You may need to scale out the replicas further, depending on the load.

Also consider running the gateway on a dedicated set of nodes in the cluster. Benefits to this approach include:

- **Isolation.** All inbound traffic goes to a fixed set of nodes, which can be isolated from backend services.
- **Stable configuration.** If the gateway is misconfigured, the entire application may become unavailable.

- Performance. You may want to use a specific VM configuration for the gateway for performance reasons.

## Next steps

The previous articles have looked at the interfaces *between* microservices or between microservices and client applications. By design, these interfaces treat each service as an opaque box. In particular, microservices should never expose implementation details about how they manage data. That has implications for data integrity and data consistency, explored in the next article.

[Data considerations for microservices](#)

## Related resources

- [Design APIs for microservices](#)
- [Design a microservices architecture](#)
- [Using domain analysis to model microservices](#)
- [Microservices assessment and readiness](#)

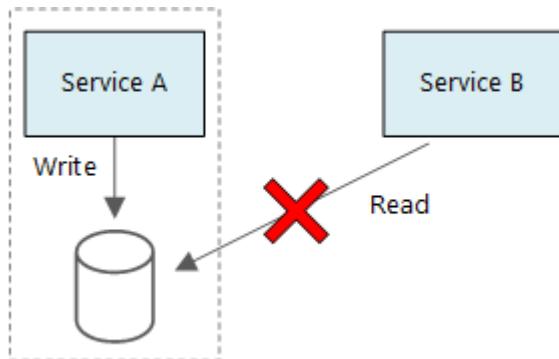
# Data considerations for microservices

Azure DevOps

This article describes considerations for managing data in a microservices architecture. Because every microservice manages its own data, data integrity and data consistency are critical challenges.

A basic principle of microservices is that each service manages its own data. Two services should not share a data store. Instead, each service is responsible for its own private data store, which other services cannot access directly.

The reason for this rule is to avoid unintentional coupling between services, which can result if services share the same underlying data schemas. If there is a change to the data schema, the change must be coordinated across every service that relies on that database. By isolating each service's data store, we can limit the scope of change, and preserve the agility of truly independent deployments. Another reason is that each microservice may have its own data models, queries, or read/write patterns. Using a shared data store limits each team's ability to optimize data storage for their particular service.



This approach naturally leads to [polyglot persistence](#) — the use of multiple data storage technologies within a single application. One service might require the schema-on-read capabilities of a document database. Another might need the referential integrity provided by an RDBMS. Each team is free to make the best choice for their service.

## ⓘ Note

It's fine for services to share the same physical database server. The problem occurs when services share the same schema, or read and write to the same set of

database tables.

## Challenges

Some challenges arise from this distributed approach to managing data. First, there may be redundancy across the data stores, with the same item of data appearing in multiple places. For example, data might be stored as part of a transaction, then stored elsewhere for analytics, reporting, or archiving. Duplicated or partitioned data can lead to issues of data integrity and consistency. When data relationships span multiple services, you can't use traditional data management techniques to enforce the relationships.

Traditional data modeling uses the rule of "one fact in one place." Every entity appears exactly once in the schema. Other entities may hold references to it but not duplicate it. The obvious advantage to the traditional approach is that updates are made in a single place, which avoids problems with data consistency. In a microservices architecture, you have to consider how updates are propagated across services, and how to manage eventual consistency when data appears in multiple places without strong consistency.

## Approaches to managing data

There is no single approach that's correct in all cases, but here are some general guidelines for managing data in a microservices architecture.

- Embrace eventual consistency where possible. Understand the places in the system where you need strong consistency or ACID transactions, and the places where eventual consistency is acceptable.
- When you need strong consistency guarantees, one service may represent the source of truth for a given entity, which is exposed through an API. Other services might hold their own copy of the data, or a subset of the data, that is eventually consistent with the master data but not considered the source of truth. For example, imagine an e-commerce system with a customer order service and a recommendation service. The recommendation service might listen to events from the order service, but if a customer requests a refund, it is the order service, not the recommendation service, that has the complete transaction history.
- For transactions, use patterns such as [Scheduler Agent Supervisor](#) and [Compensating Transaction](#) to keep data consistent across several services. You may need to store an additional piece of data that captures the state of a unit of work that spans multiple services, to avoid partial failure among multiple services. For

example, keep a work item on a durable queue while a multi-step transaction is in progress.

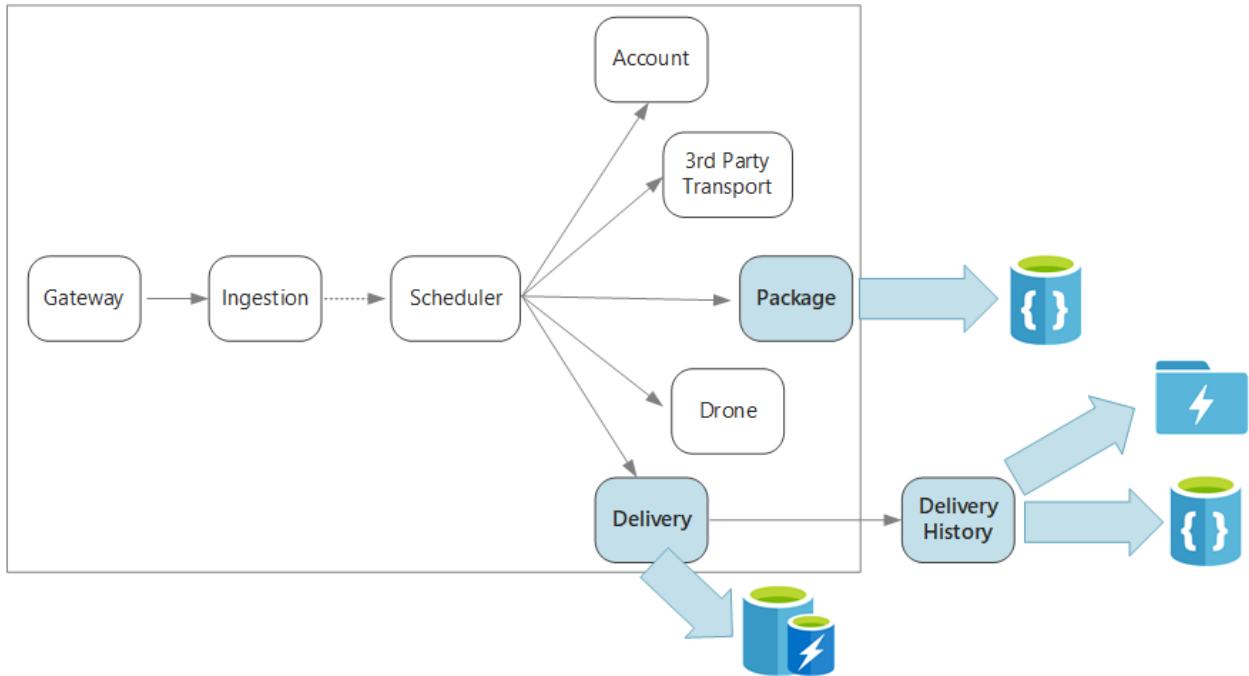
- Store only the data that a service needs. A service might only need a subset of information about a domain entity. For example, in the Shipping bounded context, we need to know which customer is associated to a particular delivery. But we don't need the customer's billing address — that's managed by the Accounts bounded context. Thinking carefully about the domain, and using a DDD approach, can help here.
- Consider whether your services are coherent and loosely coupled. If two services are continually exchanging information with each other, resulting in chatty APIs, you may need to redraw your service boundaries, by merging two services or refactoring their functionality.
- Use an [event driven architecture style](#). In this architecture style, a service publishes an event when there are changes to its public models or entities. Interested services can subscribe to these events. For example, another service could use the events to construct a materialized view of the data that is more suitable for querying.
- A service that owns events should publish a schema that can be used to automate serializing and deserializing the events, to avoid tight coupling between publishers and subscribers. Consider JSON schema or a framework like [Microsoft Bond](#), Protobuf, or Avro.
- At high scale, events can become a bottleneck on the system, so consider using aggregation or batching to reduce the total load.

## Example: Choosing data stores for the Drone Delivery application

The previous articles in this series discuss a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation [here](#). This example is ideal for the aircraft and aerospace industries.

To recap, this application defines several microservices for scheduling deliveries by drone. When a user schedules a new delivery, the client request includes information about the delivery, such as pickup and dropoff locations, and about the package, such as size and weight. This information defines a unit of work.

The various backend services care about different portions of the information in the request, and also have different read and write profiles.



## Delivery service

The Delivery service stores information about every delivery that is currently scheduled or in progress. It listens for events from the drones, and tracks the status of deliveries that are in progress. It also sends domain events with delivery status updates.

It's expected that users will frequently check the status of a delivery while they are waiting for their package. Therefore, the Delivery service requires a data store that emphasizes throughput (read and write) over long-term storage. Also, the Delivery service does not perform any complex queries or analysis, it simply fetches the latest status for a given delivery. The Delivery service team chose Azure Cache for Redis for its high read-write performance. The information stored in Redis is relatively short-lived. Once a delivery is complete, the Delivery History service is the system of record.

## Delivery History service

The Delivery History service listens for delivery status events from the Delivery service. It stores this data in long-term storage. There are two different use-cases for this historical data, which have different data storage requirements.

The first scenario is aggregating the data for the purpose of data analytics, in order to optimize the business or improve the quality of the service. Note that the Delivery History service doesn't perform the actual analysis of the data. It's only responsible for the ingestion and storage. For this scenario, the storage must be optimized for data

analysis over a large set of data, using a schema-on-read approach to accommodate a variety of data sources. [Azure Data Lake Store](#) is a good fit for this scenario. Data Lake Store is an Apache Hadoop file system compatible with Hadoop Distributed File System (HDFS), and is tuned for performance for data analytics scenarios.

The other scenario is enabling users to look up the history of a delivery after the delivery is completed. Azure Data Lake is not optimized for this scenario. For optimal performance, Microsoft recommends storing time-series data in Data Lake in folders partitioned by date. (See [Tuning Azure Data Lake Store for performance](#)). However, that structure is not optimal for looking up individual records by ID. Unless you also know the timestamp, a lookup by ID requires scanning the entire collection. Therefore, the Delivery History service also stores a subset of the historical data in Azure Cosmos DB for quicker lookup. The records don't need to stay in Azure Cosmos DB indefinitely. Older deliveries can be archived — say, after a month. This could be done by running an occasional batch process. Archiving older data can reduce costs for Cosmos DB while still keeping the data available for historical reporting from the Data Lake.

## Package service

The Package service stores information about all of the packages. The storage requirements for the Package are:

- Long-term storage.
- Able to handle a high volume of packages, requiring high write throughput.
- Support simple queries by package ID. No complex joins or requirements for referential integrity.

Because the package data is not relational, a document-oriented database is appropriate, and Azure Cosmos DB can achieve high throughput by using sharded collections. The team that works on the Package service is familiar with the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js), so they select the [MongoDB API](#) for Azure Cosmos DB. That lets them leverage their existing experience with MongoDB, while getting the benefits of Azure Cosmos DB, which is a managed Azure service.

## Next steps

Learn about design patterns that can help mitigate some common challenges in a microservices architecture.

[Design patterns for microservices](#)

## Related resources

- Using domain analysis to model microservices
- Design a microservices architecture
- Design APIs for microservices
- Microservices architecture design

# Container orchestration for microservices

Azure Kubernetes Service (AKS)

Azure Service Fabric

Azure Container Instances

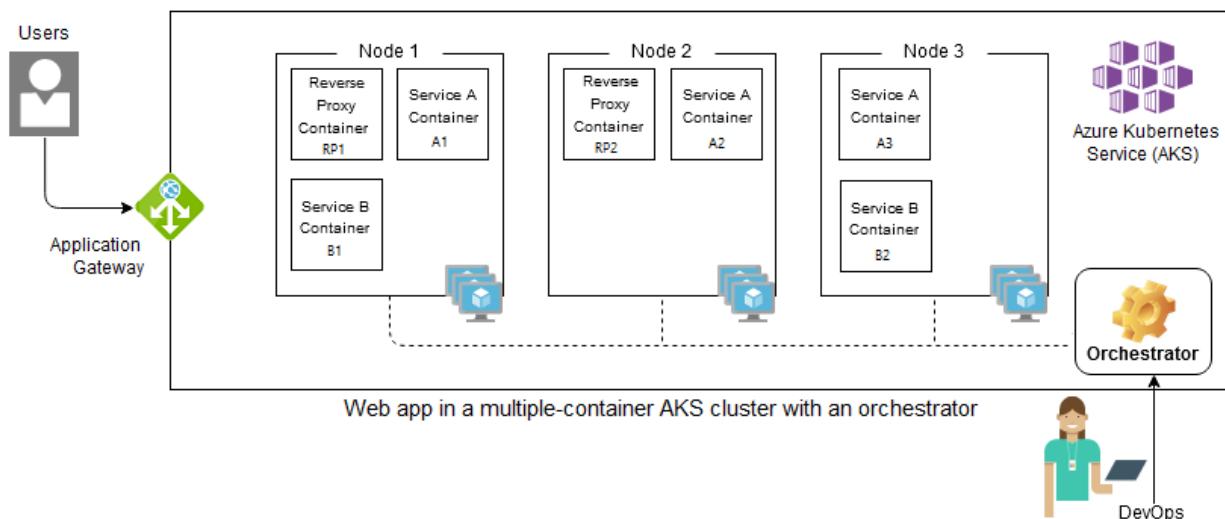
Microservices architectures typically package and deploy each microservice instance inside a single container. Many instances of the microservices might be running, each in a separate container. Containers are lightweight and short-lived, making them easy to create and destroy, but difficult to coordinate and communicate between.

This article discusses the challenges of running a containerized microservices architecture at production scale, and how container orchestration can help. The article presents several Azure container orchestration options.

## Containerized microservices architecture

In this simple containerized Azure Kubernetes Service (AKS) cluster:

- One Microservice A instance is running in Node 1, another instance in Node 2, and a third instance in Node 3.
- One instance of Microservice B is running in Node 1, and another instance in Node 3.
- Containerized [reverse proxy servers](#) are running in Nodes 1 and 2 to distribute traffic.



To manage the cluster, a DevOps team has to:

- Run multiple container instances in each node.
- Load balance traffic between the instances.
- Manage communication between dependent instances in separate nodes.
- Maintain the desired AKS cluster state.

With container orchestration, the DevOps team can represent the cluster's desired state as a configuration. A container orchestration engine enforces the desired configuration and automates all the management tasks.

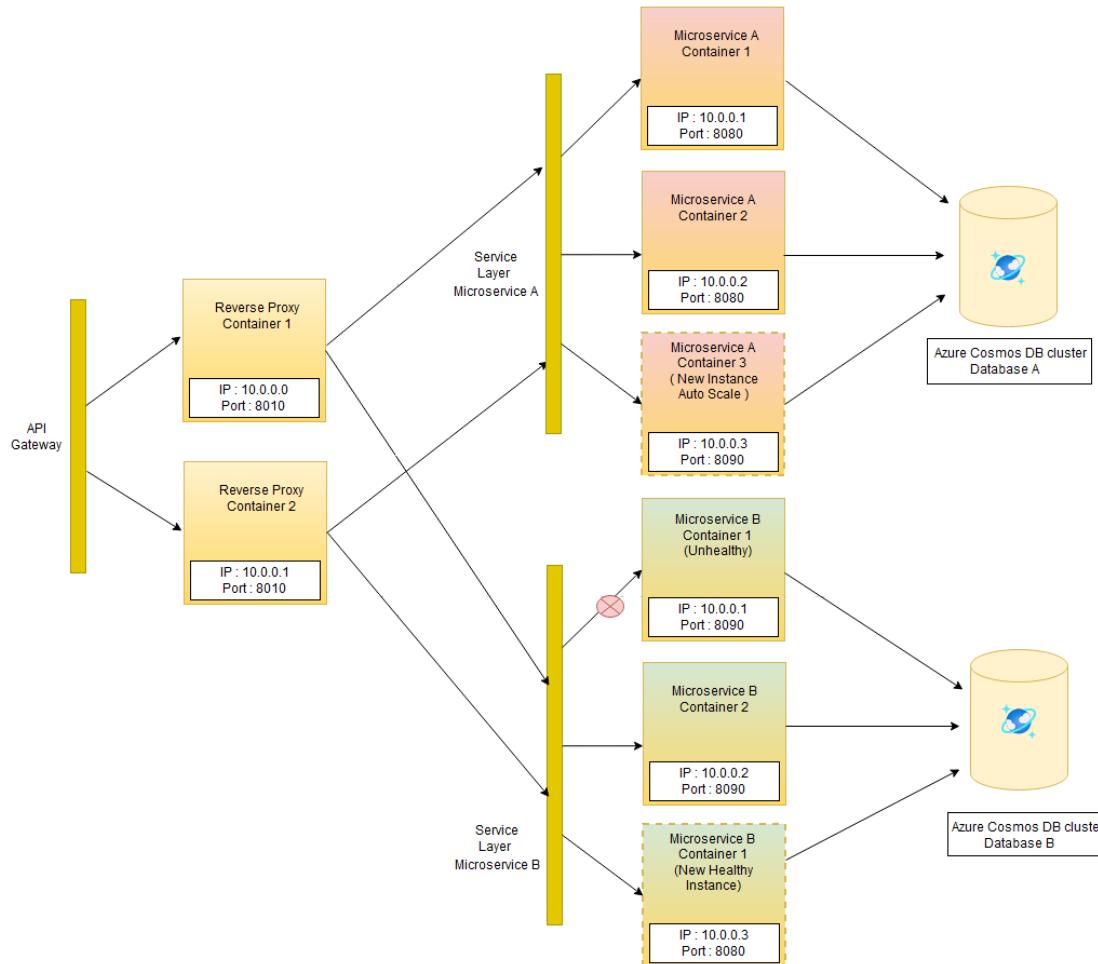
Consider containerizing a simple three-tier web application:

- A container hosts the front-end component.
- Another container hosts the middle tier or REST API layer.
- The middle tier layer communicates with a globally distributed database.

Running these containers on a single development machine might not be too hard. However, running the application in high availability mode at scale in a production cluster quickly becomes challenging. Container orchestration is crucial for large and dynamic production environments.

## Advantages of container orchestration

The following example shows how container orchestration can help manage cluster deployment, networking, and scaling.



The container orchestrator:

- Automatically scales the number of microservice instances, based on traffic or resource utilization. In the example, the orchestrator automatically adds another Microservice A instance in response to increased traffic.
- Manages the containers to reflect the configured desired state. In the example, Microservice B is configured to have two instances. One instance has become unhealthy, so the orchestrator maintains the desired state by creating another instance.
- Wraps the containers for each microservice in a simple service layer. The service layer:
  - Abstracts out complexities like IP address, port, and number of instances.
  - Load balances traffic between microservice instances.
  - Supports easy communication between dependent microservice instances.

Container orchestrators also provide flexibility and traffic control to:

- Release new versions or roll back to old versions of microservices or sets of microservices, without downtime.
- Enable side by side testing of different microservice versions.

# Choose an Azure container orchestrator

Here are some options for implementing microservices container orchestration in Azure:

- [Azure Kubernetes Service \(AKS\)](#) is a fully managed [Kubernetes](#) container orchestration service in Azure that simplifies deployment and management of containerized applications. AKS provides elastic provisioning, fast end-to-end deployment, and advanced identity and access management.
- [Azure Service Fabric](#) is a container orchestrator for deploying and managing microservices across a cluster of machines. The lightweight Service Fabric runtime supports building stateless and stateful microservices.

A key Service Fabric differentiator is its robust support for building stateful services. You can use the built-in stateful services programming model, or run containerized stateful services written in any language or code.

- [Azure Container Instances \(ACI\)](#) is the quickest and simplest way to run a container in Azure. With ACI, you don't have to manage virtual machines or adapt higher-level services.

For simple orchestration scenarios, you can use [Docker Compose](#) to define and run a multi-container application locally. Then, deploy the Docker containers as an ACI container group in a managed, serverless Azure environment. For full container orchestration scenarios, ACI can integrate with AKS to create virtual nodes for AKS orchestration.

- [Azure Spring Apps](#) is an enterprise-ready, fully managed service for [Spring Boot](#) apps. With Spring Apps, you can focus on building and running apps without having to manage infrastructure. Spring Apps comes with built-in lifecycle and orchestration management, ease of monitoring, and full integration with Azure.
- [Azure Red Hat OpenShift \(ARO\)](#) supports deployment of fully managed [OpenShift](#) clusters on Azure. Running Kubernetes production containers requires integration with frameworks and tools like image registries, storage management, monitoring, and DevOps. ARO extends Kubernetes by combining these components into a single container platform as a service (PaaS).

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Veerash Ayyagari](#) | Principal Software Engineer

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- Microservices architecture on Azure Kubernetes Service (AKS)
- Advanced Azure Kubernetes Service (AKS) microservices architecture
- CI/CD for AKS apps with Azure Pipelines
- Use API gateways in microservices
- Monitor a microservices architecture in AKS
- Microservices architecture on Azure Service Fabric
- Azure Spring Apps reference architecture

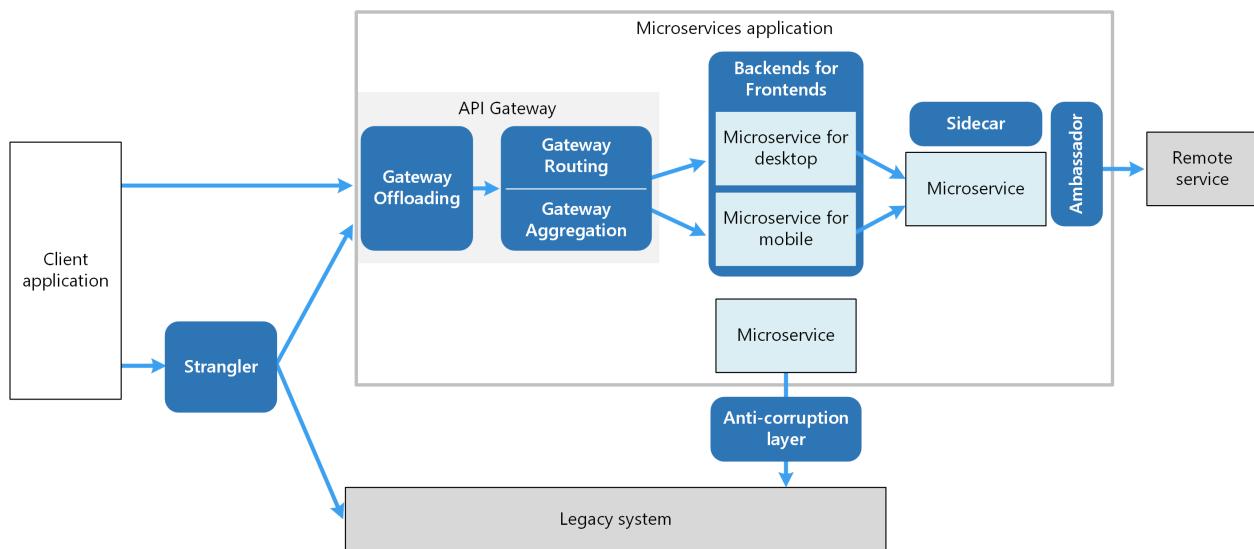
## Related resources

- Build microservices on Azure
- Design a microservices architecture
- Design patterns for microservices
- Microservices architectural style
- Azure Kubernetes Service solution journey

# Design patterns for microservices

Azure Cloud Services

The goal of microservices is to increase the velocity of application releases, by decomposing the application into small autonomous services that can be deployed independently. A microservices architecture also brings some challenges. The design patterns shown here can help mitigate these challenges.



**Ambassador** can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way. Ambassador services are often deployed as a sidecar (see below).

**Anti-corruption layer** implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.

**Backends for Frontends** creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn't need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.

**Bulkhead** isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can't consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.

**Gateway Aggregation** aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.

[Gateway Offloading](#) enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.

[Gateway Routing](#) routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.

[Messaging Bridge](#) integrates disparate systems built with different messaging infrastructures.

[Sidecar](#) deploys helper components of an application as a separate container or process to provide isolation and encapsulation.

[Strangler Fig](#) supports incremental refactoring of an application, by gradually replacing specific pieces of functionality with new services.

For the complete catalog of cloud design patterns on the Azure Architecture Center, see [Cloud Design Patterns](#).

## Next steps

- [Training: Decompose a monolithic application into a microservices architecture](#)
- [What are microservices?](#)
- [Why use a microservices approach to building applications](#)
- [Microservices architecture](#)

## Related resources

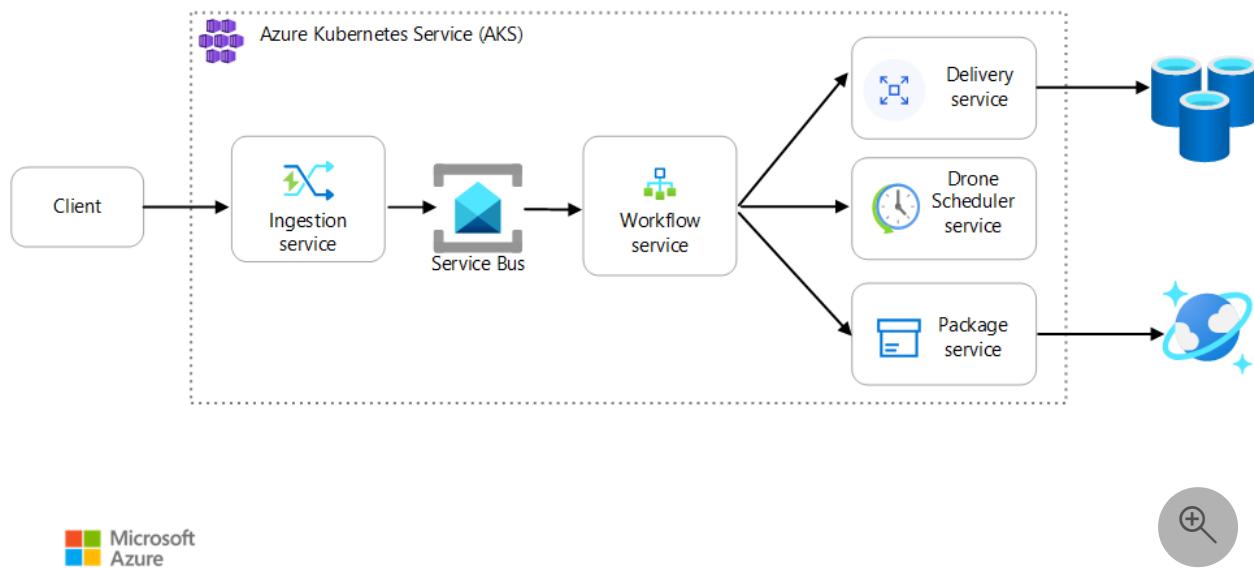
- [Microservice architecture style](#)
- [Design a microservices architecture](#)
- [Using domain analysis to model microservices](#)
- [Data considerations for microservices](#)

# Monitor a microservices application in AKS

Azure Monitor

Azure Kubernetes Service (AKS)

This article describes best practices for monitoring a microservices application that runs on Azure Kubernetes Service (AKS). Specific topics include telemetry collection, monitoring a cluster's status, metrics, logging, structured logging, and distributed tracing. The latter is illustrated in this diagram:



## Telemetry collection

In any complex application, at some point something will go wrong. In a microservices application, you need to track what's happening across dozens or even hundreds of services. To make sense of what's happening, you need to collect telemetry from the application. Telemetry can be divided into these categories: *logs*, *traces*, and *metrics*.

**Logs** are text-based records of events that occur while an application is running. They include things like application logs (trace statements) and web server logs. Logs are primarily useful for forensics and root cause analysis.

**Traces**, also called *operations*, connect the steps of a single request across multiple calls within and across microservices. They can provide structured observability into the interactions of system components. Traces can begin early in the request process, such

as within the UI of an application, and can propagate through network services across a network of microservices that handle the request.

- **Spans** are units of work within a trace. Each span is connected with a single trace and can be nested with other spans. They often correspond to individual *requests* in a cross-service operation, but they can also define work in individual components within a service. Spans also track outbound calls from one service to another. (Sometimes spans are called *dependency records*.)

**Metrics** are numerical values that can be analyzed. You can use them to observe a system in real time (or close to real time) or to analyze performance trends over time. To understand a system holistically, you need to collect metrics at various levels of the architecture, from the physical infrastructure to the application, including:

- **Node-level** metrics, including CPU, memory, network, disk, and file system usage. System metrics help you to understand resource allocation for each node in the cluster, and to troubleshoot outliers.
- **Container** metrics. For containerized applications, you need to collect metrics at the container level, not just at the VM level.
- **Application** metrics. These metrics are relevant to understanding the behavior of a service. Examples include the number of queued inbound HTTP requests, request latency, and message-queue length. Applications can also use custom metrics that are specific to the domain, like the number of business transactions processed per minute.
- **Dependent service** metrics. Services sometimes call external services or endpoints, like managed PaaS or SaaS services. Third-party services might not provide metrics. If they don't, you need to rely on your own application metrics to track statistics for latency and error rate.

## Monitoring cluster status

Use [Azure Monitor](#) to monitor the health of your clusters. The following screenshot shows a cluster that has critical errors in user-deployed pods:

Cluster Status Summary

4 1 ! 0 ! 1 ?

Total Critical Warning Unknown AKS Healthy 2 ✓  
AKS-engine Healthy 0 ✓  
Non-monitored 0

Monitored clusters (4) Non-monitored clusters (0)

Forums [Learn more](#)

Search by name...

CLUSTER NAME CLUSTER TYPE VERSION STATUS NODES USER PODS SYSTEM PODS

aksCluster-5sblgbifolai AKS 1.11.9 1 Critical 3 / 3 1 / 2 17 / 17

4 items

From here, you can drill in further to find the problem. For example, if the pod status is `ImagePullBackoff`, Kubernetes couldn't pull the container image from the registry. This problem could be caused by an invalid container tag or an authentication error during a pull from the registry.

If a container crashes, the container `State` becomes `Waiting`, with a `Reason` of `CrashLoopBackOff`. For a typical scenario, where a pod is part of a replica set and the retry policy is `Always`, this problem doesn't show as an error in the cluster status. However, you can run queries or set up alerts for this condition. For more information, see [Understand AKS cluster performance with Azure Monitor Container insights](#).

There are multiple container-specific workbooks available in the workbooks pane of an AKS resource. You can use these workbooks for a quick overview, troubleshooting, management, and insights. The following screenshot shows a list of workbooks that are available by default for AKS workloads.

^Kubernetes Services (12)

Network Policy Manager C... View your NPM metrics on your polic...	Node Disk Capacity Analyze most used disks, disk health s...	Workload Details Contains comprehensive workload inf...
Container Insights Usage Provides breakdown and details on h...	Subnet IP Usage View the usage of the IP by delegated...	Node GPU View the usage of your GPU Node an...
Node Disk IO Shows data input/output across your ...	Node Network Shows data and errors sent and receiv...	Deployments & HPAs View Deployments and HPA health an...
Kubelet Provides kubelet health, performance...	Persistent Volume Details Provides information about persistent...	Syslog Provide summary and visualisations f...

## Metrics

We recommend that you use [Monitor](#) to collect and view metrics for your AKS clusters and any other dependent Azure services.

- For cluster and container metrics, enable [Azure Monitor Container insights](#). When this feature is enabled, Monitor collects memory and processor metrics from

controllers, nodes, and containers via the Kubernetes Metrics API. For more information about the metrics that are available from Container insights, see [Understand AKS cluster performance with Azure Monitor Container insights](#).

- Use [Application Insights](#) to collect application metrics. Application Insights is an extensible application performance management (APM) service. To use it, you install an instrumentation package in your application. This package monitors the app and sends telemetry data to Application Insights. It can also pull telemetry data from the host environment. The data is then sent to Monitor. Application Insights also provides built-in correlation and dependency tracking. (See [Distributed tracing](#), later in this article.)

Application Insights has a maximum throughput that's measured in events per second, and it throttles telemetry if the data rate exceeds the limit. For details, see [Application Insights limits](#). Create different Application Insights instances for each environment, so that dev/test environments don't compete against the production telemetry for quota.

A single operation can generate many telemetry events, so if an application experiences a high volume of traffic, its telemetry capture is likely to be throttled. To mitigate this problem, you can perform sampling to reduce the telemetry traffic. The tradeoff is that your metrics will be less precise, unless the instrumentation supports [pre-aggregation](#). In that case, there will be fewer trace samples for troubleshooting, but the metrics maintain accuracy. For more information, see [Sampling in Application Insights](#). You can also reduce the data volume by pre-aggregating metrics. That is, you can calculate statistical values, like the average and standard deviation, and send those values instead of the raw telemetry. This blog post describes an approach to using Application Insights at scale: [Azure Monitoring and Analytics at Scale](#).

If your data rate is high enough to trigger throttling, and sampling or aggregation aren't acceptable, consider exporting metrics to a time-series database, like Azure Data Explorer, Prometheus, or InfluxDB, running in the cluster.

- [Azure Data Explorer](#) is an Azure-native, highly scalable data exploration service for log and telemetry data. It features support for multiple data formats, a rich query language, and connections for consuming data in popular tools like [Jupyter Notebooks](#) and [Grafana](#). Azure Data Explorer has built-in connectors to ingest log and metrics data via Azure Event Hubs. For more information, see [Ingest and query monitoring data in Azure Data Explorer](#).
- InfluxDB is a push-based system. An agent needs to push the metrics. You can use [TICK stack](#) to set up the monitoring of Kubernetes. Next, you can push metrics to InfluxDB by using [Telegraf](#), which is an agent for collecting and reporting metrics. You can use InfluxDB for irregular events and string data types.

- Prometheus is a pull-based system. It periodically scrapes metrics from configured locations. Prometheus can [scrape metrics generated by Azure Monitor](#) or `kube-state-metrics`. [kube-state-metrics](#) is a service that collects metrics from the Kubernetes API server and makes them available to Prometheus (or a scraper that's compatible with a Prometheus client endpoint). For system metrics, use [node exporter](#), which is a Prometheus exporter for system metrics. Prometheus supports floating-point data but not string data, so it's appropriate for system metrics but not logs. [Kubernetes Metrics Server](#) is a cluster-wide aggregator of resource usage data.

## Logging

Here are some of the general challenges of logging in a microservices application:

- Understanding the end-to-end processing of a client request, where multiple services might be invoked to handle a single request.
- Consolidating logs from multiple services into a single aggregated view.
- Parsing logs that come from multiple sources that use their own logging schemas or have no particular schema. Logs might be generated by third-party components that you don't control.
- Microservices architectures often generate a larger volume of logs than traditional monoliths because there are more services, network calls, and steps in a transaction. That means logging itself can be a performance or resource bottleneck for the application.

There are some additional challenges for Kubernetes-based architectures:

- Containers can move around and be rescheduled.
- Kubernetes has a networking abstraction that uses virtual IP addresses and port mappings.

In Kubernetes, the standard approach to logging is for a container to write logs to `stdout` and `stderr`. The container engine redirects these streams to a logging driver. To make querying easier, and to prevent possible loss of log data if a node stops responding, the usual approach is to collect the logs from each node and send them to a central storage location.

Azure Monitor integrates with AKS to support this approach. Monitor collects container logs and sends them to a Log Analytics workspace. From there, you can use the [Kusto Query Language](#) to write queries across the aggregated logs. For example, here's a Kusto query for showing the container logs for a specified pod:

Kusto

```
ContainerLogV2
| where PodName == "podName" //update with target pod
| project TimeGenerated, Computer, ContainerId, LogMessage, LogSource
```

Azure Monitor is a managed service, and configuring an AKS cluster to use Monitor is a simple configuration change in the CLI or Azure Resource Manager template. (For more information, see [How to enable Azure Monitor Container insights](#).) Another advantage of using Azure Monitor is that it consolidates your AKS logs with other Azure platform logs to provide a unified monitoring experience.

Azure Monitor is billed per gigabyte (GB) of data ingested into the service. (See [Azure Monitor pricing](#).) At high volumes, cost might become a consideration. There are many open-source alternatives available for the Kubernetes ecosystem. For example, many organizations use Fluentd with Elasticsearch. Fluentd is an open-source data collector, and Elasticsearch is a document database that's used for search. A challenge with these options is that they require extra configuration and management of the cluster. For a production workload, you might need to experiment with configuration settings. You'll also need to monitor the performance of the logging infrastructure.

## OpenTelemetry

OpenTelemetry is a cross-industry effort to improve tracing by standardizing the interface between applications, libraries, telemetry, and data collectors. When you use a library and framework that are instrumented with OpenTelemetry, most of the work of tracing operations that are traditionally system operations is handled by the underlying libraries, which includes the following common scenarios:

- Logging of basic request operations, like start time, exit time, and duration
- Exceptions thrown
- Context propagation (like sending a correlation ID across HTTP call boundaries)

Instead, the base libraries and frameworks that handle these operations create rich interrelated span and trace data structures and propagate them across contexts. Before OpenTelemetry, these were usually just injected as special log messages or as proprietary data structures that were specific to the vendor who built the monitoring tools. OpenTelemetry also encourages a richer instrumentation data model than a traditional logging-first approach, and the logs are more useful because the log messages are linked to the traces and spans where they were generated. This often makes finding logs that are associated with a specific operation or request easy.

Many of the Azure SDKs have been instrumented with OpenTelemetry or are in the process of implementing it.

An application developer can add manual instrumentation by using the OpenTelemetry SDKs to do the following activities:

- Add instrumentation where an underlying library doesn't provide it.
- Enrich the trace context by adding spans to expose application-specific units of work (like an order loop that creates a span for the processing of each order line).
- Enrich existing spans with entity keys to enable easier tracing. (For example, add an OrderID key/value to the request that processes that order.) These keys are surfaced by the monitoring tools as structured values for querying, filtering, and aggregating (without parsing out log message strings or looking for combinations of log message sequences, as was common with a logging-first approach).
- Propagate trace context by accessing trace and span attributes, injecting tracelds into responses and payloads, and/or reading tracelds from incoming messages, in order to create requests and spans.

Read more about instrumentation and the OpenTelemetry SDKs in the [OpenTelemetry documentation](#) ↗.

## Application Insights

Application Insights collects rich data from OpenTelemetry and its instrumentation libraries and captures it in an efficient data store to provide rich visualization and query support. The [Application Insights OpenTelemetry-based instrumentation libraries](#), for languages like .NET, Java, Node.js, and Python, make it easy to send telemetry data to Application Insights.

If you're using .NET Core, we recommend that you also consider the [Application Insights for Kubernetes](#) ↗ library. This library enriches Application Insights traces with additional information, like the container, node, pod, labels, and replica set.

Application Insights maps the OpenTelemetry context to its internal data model:

- Trace -> Operation
- Trace ID -> Operation ID
- Span -> Request or Dependency

Take the following considerations into account:

- Application Insights throttles telemetry if the data rate exceeds a maximum limit. For details, see [Application Insights limits](#). A single operation can generate several

telemetry events, so if an application experiences a high volume of traffic, it's likely to be throttled.

- Because Application Insights batches data, you can lose a batch if a process fails with an unhandled exception.
- Application Insights billing is based on data volume. For more information, see [Manage pricing and data volume in Application Insights](#).

## Structured logging

To make logs easier to parse, use structured logging when you can. When you use structured logging, the application writes logs in a structured format, like JSON, rather than outputting unstructured text strings. There are many structured logging libraries available. For example, here's a logging statement that uses the [Serilog library](#) for .NET Core:

C#

```
public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)
{
 logger.LogInformation("In Put action with delivery {Id}:
{@DeliveryInfo}", id, delivery.ToLogInfo());

 ...
}
```

Here, the call to `.LogInformation` includes an `Id` parameter and `DeliveryInfo` parameter. When you use structured logging, these values aren't interpolated into the message string. Instead, the log output looks something like this:

JSON

```
{"@t":"2019-06-13T00:57:09.9932697Z","@mt":"In Put action with delivery
{Id}: {@DeliveryInfo}","Id":"36585f2d-c1fa-4a3d-9e06-
a7f40b7d04ef","DeliveryInfo":{...}
```

This is a JSON string, where the `@t` field is a timestamp, `@mt` is the message string, and the remaining key/value pairs are the parameters. Outputting JSON format makes it easier to query the data in a structured way. For example, the following Log Analytics query, written in the [Kusto query language](#), searches for instances of this particular message from all containers named `fabrikam-delivery`:

Kusto

```
traces
| where customDimensions["Kubernetes.Container.Name"] == "fabrikam-delivery"
| where customDimensions["{OriginalFormat}"] == "In Put action with delivery {Id}: {@DeliveryInfo}"
| project message, customDimensions["Id"], customDimensions["@DeliveryInfo"]
```

If you view the result in the Azure portal, you can see that `DeliveryInfo` is a structured record that contains the serialized representation of the `DeliveryInfo` model:

message		customDimensions_Id	customDimensions_@DeliveryInfo
▼ In Put action with delivery 36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef: {"Id": "36585f...	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef	{"Id": "36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef", "Owner": "user id for logging", "AccountI...	
message	In Put action with delivery 36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef: {"Id": "36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef", "Owner": "user id for logging", "AccountI...		
customDimensions_Id	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef		
customDimensions_@DeliveryInfo	{"Id": "36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef", "Owner": "user id for logging", "AccountI...		
ConfirmationRequired	0		
Deadline	string		
DroneId	AssignedDroneId01ba4d0b-c01a-4369-ba75-51bde0e76cc9		
► Dropoff	{"Altitude": 0.31507750848078986, "Latitude": 0.753494655598651, "Longitude": 0.8935283077384942}		
Expedited	true		
Id	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef		
► Owner	{"UserId": "user id for logging", "AccountId": "52dadf0c-0067-43e7-af76-86e32b48bc5e"}		
► Pickup	{"Altitude": 0.2929516161293497, "Latitude": 0.26815900219052985, "Longitude": 0.7984184430904773}		

Here's the JSON from this example:

JSON

```
{
 "Id": "36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef",
 "Owner": {
 "UserId": "user id for logging",
 "AccountId": "52dadf0c-0067-43e7-af76-86e32b48bc5e"
 },
 "Pickup": {
 "Altitude": 0.29295161612934972,
 "Latitude": 0.26815900219052985,
 "Longitude": 0.79841844309047727
 },
 "Dropoff": {
 "Altitude": 0.31507750848078986,
 "Latitude": 0.753494655598651,
 "Longitude": 0.89352830773849423
 },
 "Deadline": "string",
 "Expedited": true,
 "ConfirmationRequired": 0,
 "DroneId": "AssignedDroneId01ba4d0b-c01a-4369-ba75-51bde0e76cc9"
}
```

Many log messages mark the start or end of a unit of work, or they connect a business entity with a set of messages and operations for traceability. In many cases, enriching OpenTelemetry span and request objects is a better approach than only logging the start and end of the operation. Doing so adds that context to all connected traces and child operations, and it puts that information in the scope of the full operation. The OpenTelemetry SDKs for various languages support creating spans or adding custom attributes on spans. For example, the following code [uses the Java OpenTelemetry SDK](#), which is [supported by Application Insights](#). An existing parent span (for example, a request span that's associated with a REST controller call and created by the web framework being used) can be enriched with an entity ID that's associated with it, as shown here:

Java

```
import io.opentelemetry.api.trace.Span;
// ...
Span.current().setAttribute("A1234", deliveryId);
```

This code sets a key or value on the current span, which is connected to operations and log messages that occur under that span. The value appears in the Application Insights request object, as shown here:

Kusto

```
requests
| extend deliveryId = tostring(customDimensions.deliveryId) // promote to
column value (optional)
| where deliveryId == "A1234"
| project timestamp, name, url, success, resultCode, duration, operation_Id,
deliveryId
```

This technique becomes more powerful when used with logs, filtering, and annotating log traces with span context, as shown here:

Kusto

```
requests
| extend deliveryId = tostring(customDimensions.deliveryId) // promote to
column value (optional)
| where deliveryId == "A1234"
| project deliveryId, operation_Id, requestTimestamp = timestamp,
requestDuration = duration // keep some request info
| join kind=inner traces on operation_Id // join logs only for this
deliveryId
```

```
| project requestTimestamp, requestDuration, logTimestamp = timestamp,
deliveryId, message
```

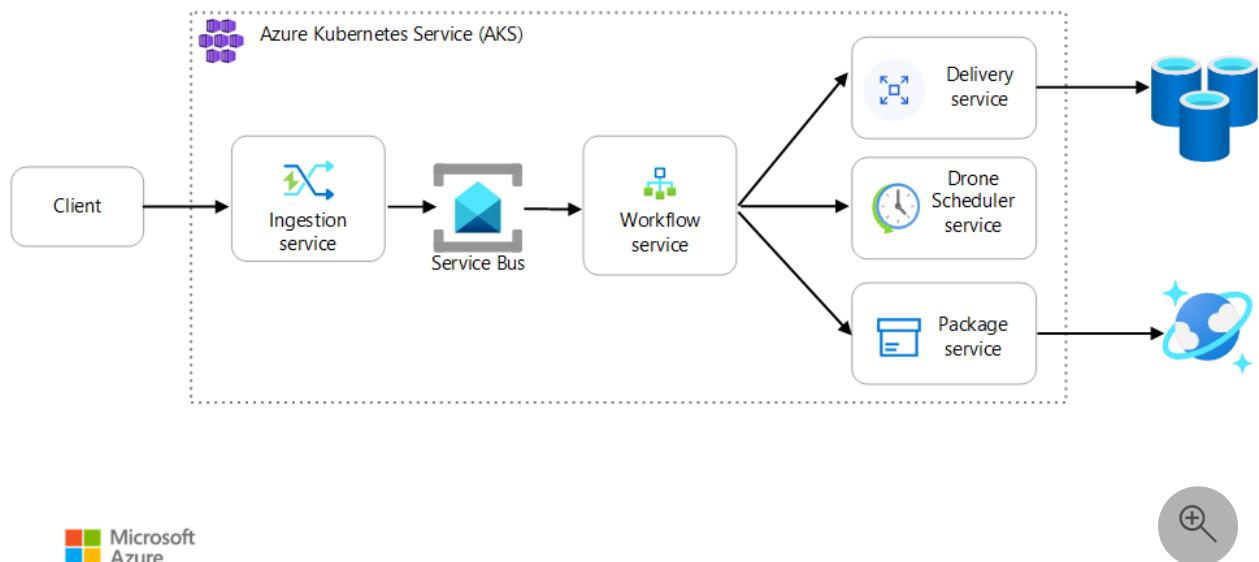
If you use a library or framework that's already instrumented with OpenTelemetry, it handles creating spans and requests, but the application code might also create units of work. For example, a method that loops through an array of entities that performs work on each one might create a span for each iteration of the processing loop. For information about adding instrumentation to application and library code, see the [OpenTelemetry instrumentation documentation](#) ↗.

## Distributed tracing

One of the challenges when you use microservices is understanding the flow of events across services. A single transaction can involve calls to multiple services.

### Example of distributed tracing

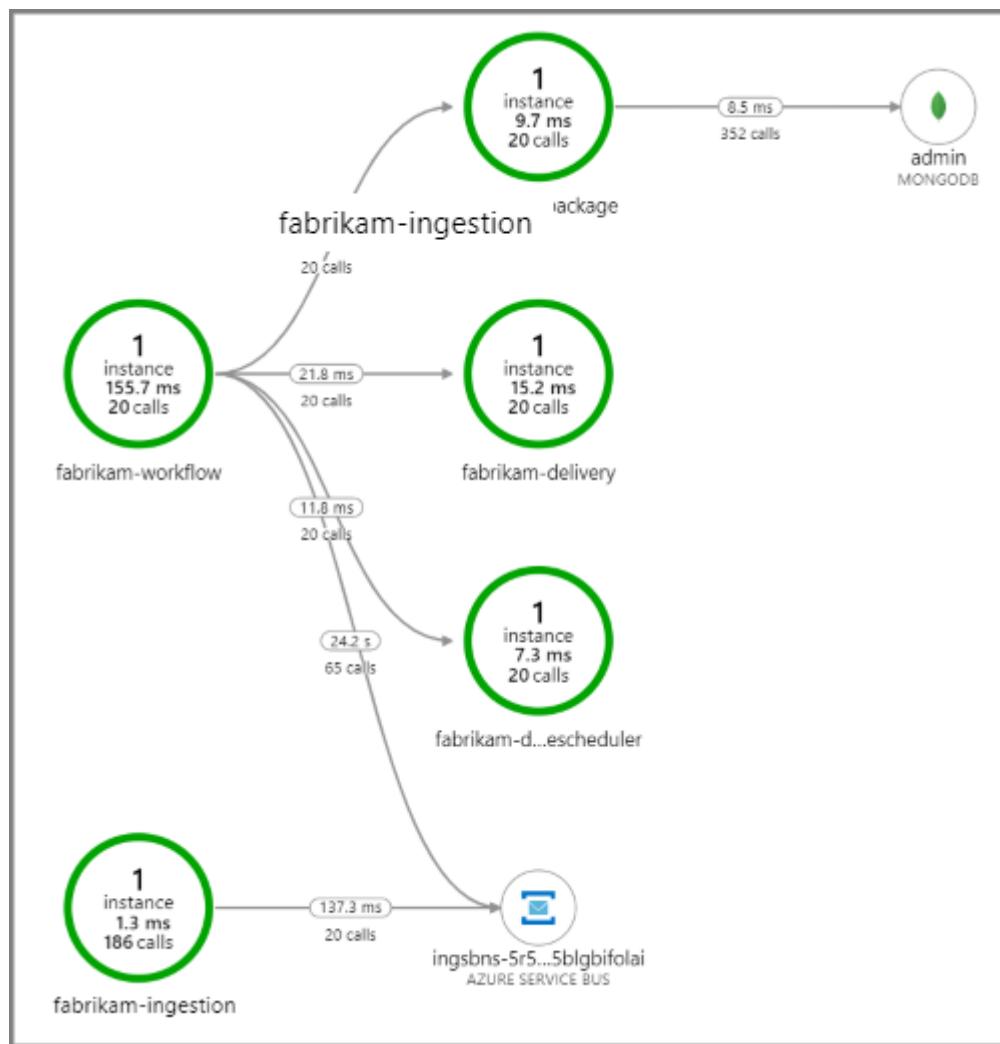
This example describes the path of a distributed transaction through a set of microservices. The example is based on a [drone delivery application](#).



In this scenario, the distributed transaction includes these steps:

1. The Ingestion service puts a message on an Azure Service Bus queue.
2. The Workflow service pulls the message from the queue.
3. The Workflow service calls three back-end services to process the request (Drone Scheduler, Package, and Delivery).

The following screenshot shows the [application map](#) for the drone delivery application. This map shows calls to the public API endpoint that result in a workflow that involves five microservices.



The arrows from `fabrikam-workflow` and `fabrikam-ingestion` to a Service Bus queue show where the messages are sent and received. You can't tell from the diagram which service is sending messages and which is receiving. The arrows just show that both services are calling Service Bus. But information about which service is sending and which is receiving is available in the details:

fabrikam-wor... → ingsbns-5r5bl... AZURE SERVICE BUS

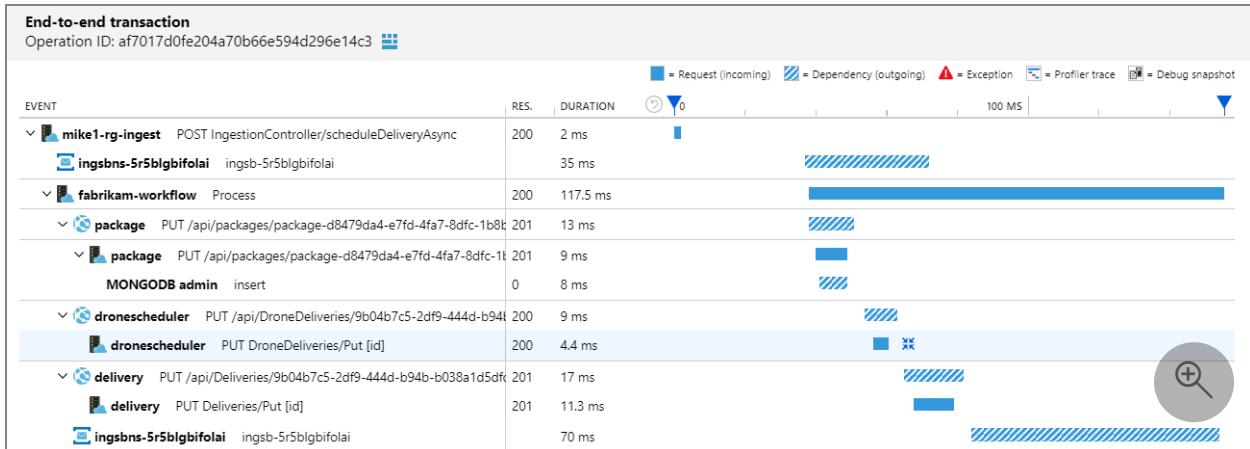
View in Analytics

SLOWEST CALLS BY NAME

NAME	DURATION (AVG)
Receive	58 s
Complete	87.1 ms

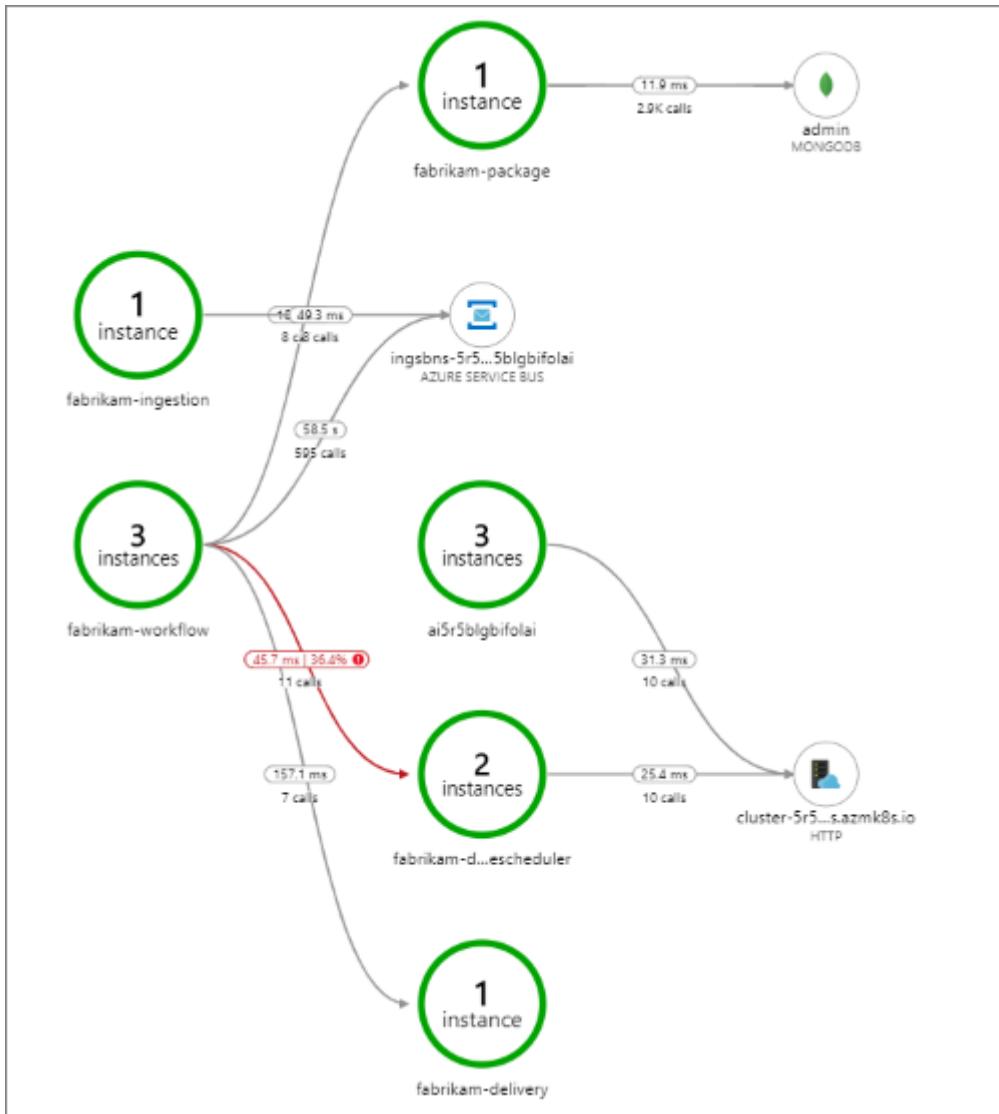
**Investigate performance**

Because every call includes an operation ID, you can also view the end-to-end steps of a single transaction, including timing information and the HTTP calls at each step. Here's the visualization of one such transaction:

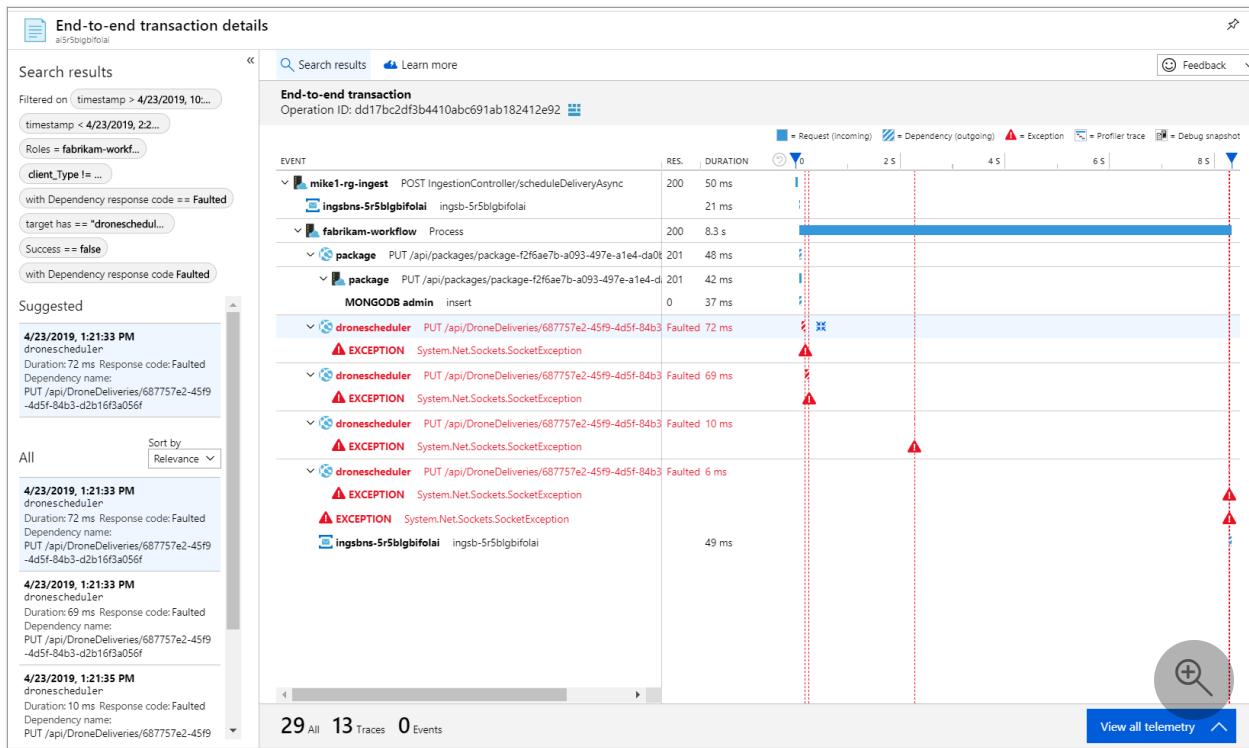


This visualization shows the steps from the ingestion service to the queue, from the queue to the Workflow service, and from the Workflow service to the other back-end services. The last step is the Workflow service marking the Service Bus message as completed.

This example shows calls to a back-end service that are failing:



This map shows that a large fraction (36%) of calls to the Drone Scheduler service failed during the period of the query. The end-to-end transaction view reveals that an exception occurs when an HTTP PUT request is sent to the service:



If you drill in further, you can see that the exception is a socket exception: "No such device or address."

```
Fabrikam.Workflow.Service.Services.BackendServiceCallFailedException:
No such device or address
---u003e System.Net.Http.HttpRequestException: No such device or address
---u003e System.Net.Sockets.SocketException: No such device or address
```

This exception suggests that the back-end service isn't reachable. At this point, you might use `kubectl` to view the deployment configuration. In this example, the service host name isn't resolving because of an error in the Kubernetes configuration files. The article [Debug Services](#) in the Kubernetes documentation has tips for diagnosing this type of error.

Here are some common causes of errors:

- Code bugs. These bugs might appear as:
  - Exceptions. Look in the Application Insights logs to view the exception details.
  - A process failing. Look at container and pod status, and view container logs or Application Insights traces.
  - HTTP 5xx errors.
- Resource exhaustion:
  - Look for throttling (HTTP 429) or request timeouts.
  - Examine container metrics for CPU, memory, and disk.
  - Look at the configurations for container and pod resource limits.

- Service discovery. Examine the Kubernetes service configuration and port mappings.
- API mismatch. Look for HTTP 400 errors. If APIs are versioned, look at the version that's being called.
- Error in pulling a container image. Look at the pod specification. Also make sure that the cluster is authorized to pull from the container registry.
- RBAC problems.

## Next steps

Learn more about features in Azure Monitor that support monitoring of applications on AKS:

- [Azure Monitor container insights overview](#)
- [Understand AKS cluster performance with Azure Monitor Container insights](#)

## Related resources

- [Performance tuning a distributed application](#)
- [Using domain analysis to model microservices](#)
- [Design a microservices architecture](#)
- [Design APIs for microservices](#)

# CI/CD for microservices architectures

Azure

Faster release cycles are one of the major advantages of microservices architectures. But without a good CI/CD process, you won't achieve the agility that microservices promise. This article describes the challenges and recommends some approaches to the problem.

## What is CI/CD?

When we talk about CI/CD, we're really talking about several related processes: Continuous integration, continuous delivery, and continuous deployment.

- **Continuous integration.** Code changes are frequently merged into the main branch. Automated build and test processes ensure that code in the main branch is always production-quality.
- **Continuous delivery.** Any code changes that pass the CI process are automatically published to a production-like environment. Deployment into the live production environment may require manual approval, but is otherwise automated. The goal is that your code should always be *ready* to deploy into production.
- **Continuous deployment.** Code changes that pass the previous two steps are automatically deployed *into production*.

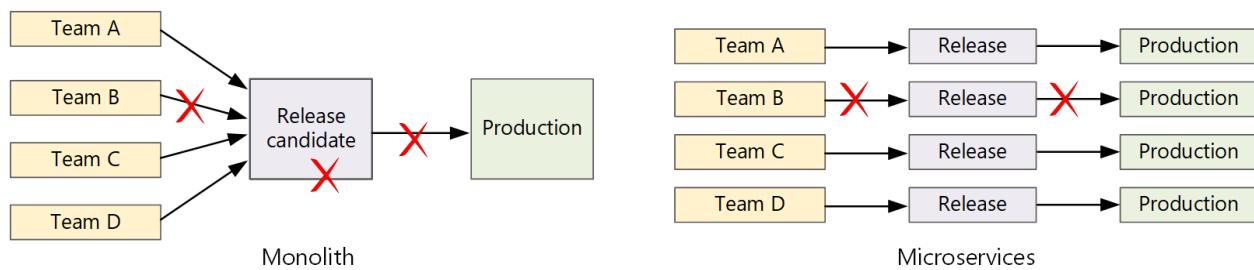
Here are some goals of a robust CI/CD process for a microservices architecture:

- Each team can build and deploy the services that it owns independently, without affecting or disrupting other teams.
- Before a new version of a service is deployed to production, it gets deployed to dev/test/QA environments for validation. Quality gates are enforced at each stage.
- A new version of a service can be deployed side by side with the previous version.
- Sufficient access control policies are in place.
- For containerized workloads, you can trust the container images that are deployed to production.

# Why a robust CI/CD pipeline matters

In a traditional monolithic application, there is a single build pipeline whose output is the application executable. All development work feeds into this pipeline. If a high-priority bug is found, a fix must be integrated, tested, and published, which can delay the release of new features. You can mitigate these problems by having well-factored modules and using feature branches to minimize the impact of code changes. But as the application grows more complex, and more features are added, the release process for a monolith tends to become more brittle and likely to break.

Following the microservices philosophy, there should never be a long release train where every team has to get in line. The team that builds service "A" can release an update at any time, without waiting for changes in service "B" to be merged, tested, and deployed.



To achieve a high release velocity, your release pipeline must be automated and highly reliable to minimize risk. If you release to production one or more times daily, regressions or service disruptions must be rare. At the same time, if a bad update does get deployed, you must have a reliable way to quickly roll back or roll forward to a previous version of a service.

## Challenges

- **Many small independent code bases.** Each team is responsible for building its own service, with its own build pipeline. In some organizations, teams may use separate code repositories. Separate repositories can lead to a situation where the knowledge of how to build the system is spread across teams, and nobody in the organization knows how to deploy the entire application. For example, what happens in a disaster recovery scenario, if you need to quickly deploy to a new cluster?

**Mitigation:** Have a unified and automated pipeline to build and deploy services, so that this knowledge is not "hidden" within each team.

- **Multiple languages and frameworks.** With each team using its own mix of technologies, it can be difficult to create a single build process that works across the organization. The build process must be flexible enough that every team can adapt it for their choice of language or framework.

**Mitigation:** Containerize the build process for each service. That way, the build system just needs to be able to run the containers.

- **Integration and load testing.** With teams releasing updates at their own pace, it can be challenging to design robust end-to-end testing, especially when services have dependencies on other services. Moreover, running a full production cluster can be expensive, so it's unlikely that every team will run its own full cluster at production scales, just for testing.
- **Release management.** Every team should be able to deploy an update to production. That doesn't mean that every team member has permissions to do so. But having a centralized Release Manager role can reduce the velocity of deployments.

**Mitigation:** The more that your CI/CD process is automated and reliable, the less there should be a need for a central authority. That said, you might have different policies for releasing major feature updates versus minor bug fixes. Being decentralized doesn't mean zero governance.

- **Service updates.** When you update a service to a new version, it shouldn't break other services that depend on it.

**Mitigation:** Use deployment techniques such as blue-green or canary release for non-breaking changes. For breaking API changes, deploy the new version side by side with the previous version. That way, services that consume the previous API can be updated and tested for the new API. See [Updating services](#), below.

## Monorepo vs. multi-repo

Before creating a CI/CD workflow, you must know how the code base will be structured and managed.

- Do teams work in separate repositories or in a monorepo (single repository)?
- What is your branching strategy?
- Who can push code to production? Is there a release manager role?

The monorepo approach has been gaining favor but there are advantages and disadvantages to both.

	Monorepo	Multiple repos
Advantages	Code sharing Easier to standardize code and tooling Easier to refactor code Discoverability - single view of the code	Clear ownership per team Potentially fewer merge conflicts Helps to enforce decoupling of microservices
Challenges	Changes to shared code can affect multiple microservices Greater potential for merge conflicts Tooling must scale to a large code base Access control More complex deployment process	Harder to share code Harder to enforce coding standards Dependency management Diffuse code base, poor discoverability Lack of shared infrastructure

## Updating services

There are various strategies for updating a service that's already in production. Here we discuss three common options: Rolling update, blue-green deployment, and canary release.

### Rolling updates

In a rolling update, you deploy new instances of a service, and the new instances start receiving requests right away. As the new instances come up, the previous instances are removed.

**Example.** In Kubernetes, rolling updates are the default behavior when you update the pod spec for a [Deployment](#). The Deployment controller creates a new ReplicaSet for the updated pods. Then it scales up the new ReplicaSet while scaling down the old one, to maintain the desired replica count. It doesn't delete old pods until the new ones are ready. Kubernetes keeps a history of the update, so you can roll back an update if needed.

**Example.** Azure Service Fabric uses the rolling update strategy by default. This strategy is best suited for deploying a version of a service with new features without changing existing APIs. Service Fabric starts an upgrade deployment by updating the application

type to a subset of the nodes or an update domain. It then rolls forward to the next update domain until all domains are upgraded. If an upgrade domain fails to update, the application type rolls back to the previous version across all domains. Be aware that an application type with multiple services (and if all services are updated as part of one upgrade deployment) is prone to failure. If one service fails to update, the entire application is rolled back to the previous version and the other services are not updated.

One challenge of rolling updates is that during the update process, a mix of old and new versions are running and receiving traffic. During this period, any request could get routed to either of the two versions.

For breaking API changes, a good practice is to support both versions side by side, until all clients of the previous version are updated. See [API versioning](#).

## Blue-green deployment

In a blue-green deployment, you deploy the new version alongside the previous version. After you validate the new version, you switch all traffic at once from the previous version to the new version. After the switch, you monitor the application for any problems. If something goes wrong, you can swap back to the old version. Assuming there are no problems, you can delete the old version.

With a more traditional monolithic or N-tier application, blue-green deployment generally meant provisioning two identical environments. You would deploy the new version to a staging environment, then redirect client traffic to the staging environment — for example, by swapping VIP addresses. In a microservices architecture, updates happen at the microservice level, so you would typically deploy the update into the same environment and use a service discovery mechanism to swap.

**Example.** In Kubernetes, you don't need to provision a separate cluster to do blue-green deployments. Instead, you can take advantage of selectors. Create a new [Deployment](#) resource with a new pod spec and a different set of labels. Create this deployment, without deleting the previous deployment or modifying the service that points to it. Once the new pods are running, you can update the service's selector to match the new deployment.

One drawback of blue-green deployment is that during the update, you are running twice as many pods for the service (current and next). If the pods require a lot of CPU or memory resources, you may need to scale out the cluster temporarily to handle the resource consumption.

## Canary release

In a canary release, you roll out an updated version to a small number of clients. Then you monitor the behavior of the new service before rolling it out to all clients. This lets you do a slow rollout in a controlled fashion, observe real data, and spot problems before all customers are affected.

A canary release is more complex to manage than either blue-green or rolling update, because you must dynamically route requests to different versions of the service.

**Example.** In Kubernetes, you can configure a [Service](#) to span two replica sets (one for each version) and adjust the replica counts manually. However, this approach is rather coarse-grained, because of the way Kubernetes load balances across pods. For example, if you have a total of 10 replicas, you can only shift traffic in 10% increments. If you are using a service mesh, you can use the service mesh routing rules to implement a more sophisticated canary release strategy.

## Next steps

- [Learning path: Define and implement continuous integration](#)
- [Training: Introduction to continuous delivery](#)
- [Microservices architecture](#)
- [Why use a microservices approach to building applications](#)

## Related resources

- [CI/CD for microservices on Kubernetes](#)
- [Design a microservices architecture](#)
- [Using domain analysis to model microservices](#)
- [Monitor a microservices architecture in Azure Kubernetes Service \(AKS\)](#)

# Build a CI/CD pipeline for microservices on Kubernetes with Azure DevOps and Helm

Azure Kubernetes Service (AKS)

Azure Container Registry

Azure DevOps

It can be challenging to create a reliable continuous integration/continuous delivery (CI/CD) process for a microservices architecture. Individual teams must be able to release services quickly and reliably, without disrupting other teams or destabilizing the application as a whole.

This article describes an example CI/CD pipeline for deploying microservices to Azure Kubernetes Service (AKS). Every team and project is different, so don't take this article as a set of hard-and-fast rules. Instead, it's meant to be a starting point for designing your own CI/CD process.

The goals of a CI/CD pipeline for Kubernetes hosted microservices can be summarized as follows:

- Teams can build and deploy their services independently.
- Code changes that pass the CI process are automatically deployed to a production-like environment.
- Quality gates are enforced at each stage of the pipeline.
- A new version of a service can be deployed side by side with the previous version.

For more background, see [CI/CD for microservices architectures](#).

## Assumptions

For purposes of this example, here are some assumptions about the development team and the code base:

- The code repository is a monorepo, with folders organized by microservice.
- The team's branching strategy is based on [trunk-based development](#).
- The team uses [release branches](#) to manage releases. Separate releases are created for each microservice.
- The CI/CD process uses [Azure Pipelines](#) to build, test, and deploy the microservices to AKS.
- The container images for each microservice are stored in [Azure Container Registry](#).

- The team uses Helm charts to package each microservice.
- A push deployment model is used, where Azure Pipelines and associated agents perform deployments by connecting directly to the AKS cluster.

These assumptions drive many of the specific details of the CI/CD pipeline. However, the basic approach described here can be adapted for other processes, tools, and services, such as Jenkins or Docker Hub.

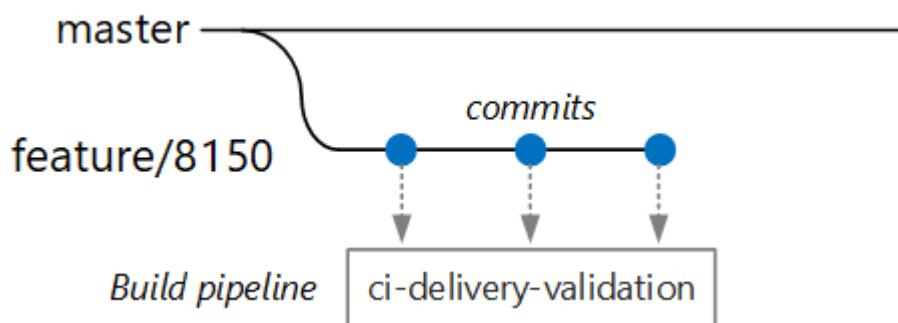
## Alternatives

The following are common alternatives customers might use when choosing a CI/CD strategy with Azure Kubernetes Service:

- As an alternative to using Helm as a package management and deployment tool, [Kustomize](#) is a Kubernetes native configuration management tool that introduces a template-free way to customize and parameterize application configuration.
- As an alternative to using Azure DevOps for Git repositories and pipelines, [GitHub Repositories](#) can be used for private and public Git repositories, and [GitHub Actions](#) can be used for CI/CD pipelines.
- As an alternative to using a push deployment model, managing Kubernetes configuration at large scale can be done using [GitOps \(pull deployment model\)](#), where an in-cluster Kubernetes operator synchronizes cluster state, based on the configuration that's stored in a Git repository.

## Validation builds

Suppose that a developer is working on a microservice called the Delivery Service. While developing a new feature, the developer checks code into a feature branch. By convention, feature branches are named `feature/*`.



The build definition file includes a trigger that filters by the branch name and the source path:

## YAML

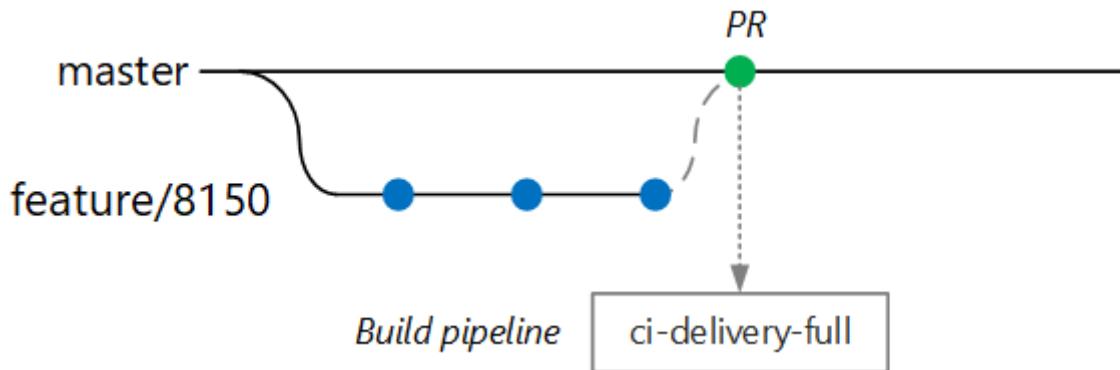
```
trigger:
 batch: true
 branches:
 include:
 # for new release to production: release flow strategy
 - release/delivery/v*
 - refs/release/delivery/v*
 - master
 - feature/delivery/*
 - topic/delivery/*
 paths:
 include:
 - /src/shipping/delivery/
```

Using this approach, each team can have its own build pipeline. Only code that is checked into the `/src/shipping/delivery` folder triggers a build of the Delivery Service. Pushing commits to a branch that matches the filter triggers a CI build. At this point in the workflow, the CI build runs some minimal code verification:

1. Build the code.
2. Run unit tests.

The goal is to keep build times short so that the developer can get quick feedback. Once the feature is ready to merge into master, the developer opens a PR. This operation triggers another CI build that performs some additional checks:

1. Build the code.
2. Run unit tests.
3. Build the runtime container image.
4. Run vulnerability scans on the image.

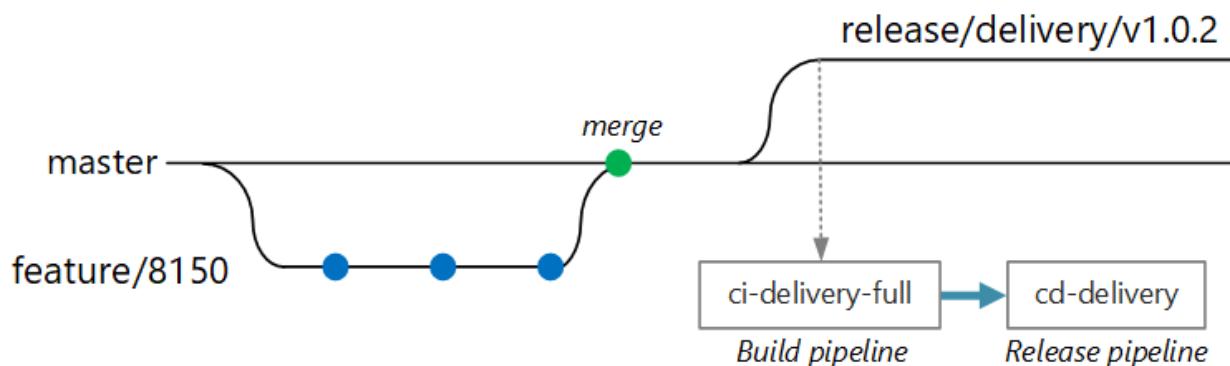


## ⓘ Note

In Azure DevOps Repos, you can define policies to protect branches. For example, the policy could require a successful CI build plus a sign-off from an approver in order to merge into master.

## Full CI/CD build

At some point, the team is ready to deploy a new version of the Delivery service. The release manager creates a branch from the main branch with this naming pattern: `release/<microservice name>/<semver>`. For example, `release/delivery/v1.0.2`.



The creation of this branch triggers a full CI build that runs all of the previous steps plus:

1. Push the container image to Azure Container Registry. The image is tagged with the version number taken from the branch name.
2. Run `helm package` to package the Helm chart for the service. The chart is also tagged with a version number.
3. Push the Helm package to Container Registry.

Assuming this build succeeds, it triggers a deployment (CD) process using an Azure Pipelines [release pipeline](#). This pipeline has the following steps:

1. Deploy the Helm chart to a QA environment.
2. An approver signs off before the package moves to production. See [Release deployment control using approvals](#).
3. Retag the Docker image for the production namespace in Azure Container Registry. For example, if the current tag is `myrepo.azurecr.io/delivery:v1.0.2`, the production tag is `myrepo.azurecr.io/prod/delivery:v1.0.2`.
4. Deploy the Helm chart to the production environment.

Even in a monorepo, these tasks can be scoped to individual microservices so that teams can deploy with high velocity. The process has some manual steps: Approving

PRs, creating release branches, and approving deployments into the production cluster. These steps are manual; they could be automated if the organization prefers.

## Isolation of environments

You will have multiple environments where you deploy services, including environments for development, smoke testing, integration testing, load testing, and finally, production. These environments need some level of isolation. In Kubernetes, you have a choice between physical isolation and logical isolation. Physical isolation means deploying to separate clusters. Logical isolation uses namespaces and policies, as described earlier.

Our recommendation is to create a dedicated production cluster along with a separate cluster for your dev/test environments. Use logical isolation to separate environments within the dev/test cluster. Services deployed to the dev/test cluster should never have access to data stores that hold business data.

## Build process

When possible, package your build process into a Docker container. This configuration allows you to build code artifacts using Docker and without configuring a build environment on each build machine. A containerized build process makes it easy to scale out the CI pipeline by adding new build agents. Also, any developer on the team can build the code simply by running the build container.

By using multi-stage builds in Docker, you can define the build environment and the runtime image in a single Dockerfile. For example, here's a Dockerfile that builds a .NET application:

### Dockerfile

```
FROM mcr.microsoft.com/dotnet/core/runtime:3.1 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src/Fabrikam.Workflow.Service

COPY Fabrikam.Workflow.Service/Fabrikam.Workflow.Service.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.csproj

COPY Fabrikam.Workflow.Service/..
RUN dotnet build Fabrikam.Workflow.Service.csproj -c release -o /app --no-restore

FROM build AS testrunner
WORKDIR /src/tests
```

```
COPY Fabrikam.Workflow.Service.Tests/*.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.Tests.csproj

COPY Fabrikam.Workflow.Service.Tests/..
ENTRYPOINT ["dotnet", "test", "--logger:trx"]

FROM build AS publish
RUN dotnet publish Fabrikam.Workflow.Service.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Fabrikam.Workflow.Service.dll"]
```

This Dockerfile defines several build stages. Notice that the stage named `base` uses the .NET runtime, while the stage named `build` uses the full .NET SDK. The `build` stage is used to build the .NET project. But the final runtime container is built from `base`, which contains just the runtime and is significantly smaller than the full SDK image.

## Building a test runner

Another good practice is to run unit tests in the container. For example, here is part of a Docker file that builds a test runner:

```
Dockerfile

FROM build AS testrunner
WORKDIR /src/tests

COPY Fabrikam.Workflow.Service.Tests/*.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.Tests.csproj

COPY Fabrikam.Workflow.Service.Tests/..
ENTRYPOINT ["dotnet", "test", "--logger:trx"]
```

A developer can use this Docker file to run the tests locally:

```
Bash

docker build . -t delivery-test:1 --target=testrunner
docker run delivery-test:1
```

The CI pipeline should also run the tests as part of the build verification step.

Note that this file uses the Docker `ENTRYPOINT` command to run the tests, not the Docker `RUN` command.

- If you use the `RUN` command, the tests run every time you build the image. By using `ENTRYPOINT`, the tests are opt-in. They run only when you explicitly target the `testrunner` stage.
- A failing test doesn't cause the Docker `build` command to fail. That way, you can distinguish container build failures from test failures.
- Test results can be saved to a mounted volume.

## Container best practices

Here are some other best practices to consider for containers:

- Define organization-wide conventions for container tags, versioning, and naming conventions for resources deployed to the cluster (pods, services, and so on). That can make it easier to diagnose deployment issues.
- During the development and test cycle, the CI/CD process will build many container images. Only some of those images are candidates for release, and then only some of those release candidates will get promoted to production. Have a clear versioning strategy so that you know which images are currently deployed to production and to help roll back to a previous version if necessary.
- Always deploy specific container version tags, not `latest`.
- Use [namespaces](#) in Azure Container Registry to isolate images that are approved for production from images that are still being tested. Don't move an image into the production namespace until you're ready to deploy it into production. If you combine this practice with semantic versioning of container images, it can reduce the chance of accidentally deploying a version that wasn't approved for release.
- Follow the principle of least privilege by running containers as a nonprivileged user. In Kubernetes, you can create a pod security policy that prevents containers from running as `root`.

## Helm charts

Consider using Helm to manage building and deploying services. Here are some of the features of Helm that help with CI/CD:

- Often, a single microservice is defined by multiple Kubernetes objects. Helm allows these objects to be packaged into a single Helm chart.
- A chart can be deployed with a single Helm command rather than a series of `kubectl` commands.

- Charts are explicitly versioned. Use Helm to release a version, view releases, and roll back to a previous version. Tracking updates and revisions, using semantic versioning, along with the ability to roll back to a previous version.
- Helm charts use templates to avoid duplicating information, such as labels and selectors, across many files.
- Helm can manage dependencies between charts.
- Charts can be stored in a Helm repository, such as Azure Container Registry, and integrated into the build pipeline.

For more information about using Container Registry as a Helm repository, see [Use Azure Container Registry as a Helm repository for your application charts](#).

A single microservice may involve multiple Kubernetes configuration files. Updating a service can mean touching all of these files to update selectors, labels, and image tags. Helm treats these as a single package called a chart and allows you to easily update the YAML files by using variables. Helm uses a template language (based on Go templates) to let you write parameterized YAML configuration files.

For example, here's part of a YAML file that defines a deployment:

```
YAML

apiVersion: apps/v1
kind: Deployment
metadata:
 name: {{ include "package.fullname" . | replace "." "" }}
 labels:
 app.kubernetes.io/name: {{ include "package.name" . }}
 app.kubernetes.io/instance: {{ .Release.Name }}
 annotations:
 kubernetes.io/change-cause: {{ .Values.reason }}
 ...
spec:
 containers:
 - name: &package-container_name_fabrikam-package
 image: {{ .Values.dockerregistry }}/{{ .Values.image.repository }}:
{{ .Values.image.tag }}
 imagePullPolicy: {{ .Values.image.pullPolicy }}
 env:
 - name: LOG_LEVEL
 value: {{ .Values.log.level }}
```

You can see that the deployment name, labels, and container spec all use template parameters, which are provided at deployment time. For example, from the command line:

Bash

```
helm install $HELM_CHARTS/package/ \
 --set image.tag=0.1.0 \
 --set image.repository=package \
 --set dockerregistry=$ACR_SERVER \
 --namespace backend \
 --name package-v0.1.0
```

Although your CI/CD pipeline could install a chart directly to Kubernetes, we recommend creating a chart archive (.tgz file) and pushing the chart to a Helm repository such as Azure Container Registry. For more information, see [Package Docker-based apps in Helm charts in Azure Pipelines](#).

## Revisions

Helm charts always have a version number, which must use [semantic versioning](#). A chart can also have an `appVersion`. This field is optional and doesn't have to be related to the chart version. Some teams might want to application versions separately from updates to the charts. But a simpler approach is to use one version number, so there's a 1:1 relation between chart version and application version. That way, you can store one chart per release and easily deploy the desired release:

Bash

```
helm install <package-chart-name> --version <desiredVersion>
```

Another good practice is to provide a change-cause annotation in the deployment template:

YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: {{ include "delivery.fullname" . | replace "." " " }}
 labels:
 ...
 annotations:
 kubernetes.io/change-cause: {{ .Values.reason }}
```

This lets you view the change-cause field for each revision, using the `kubectl rollout history` command. In the previous example, the change-cause is provided as a Helm chart parameter.

Bash

```
kubectl rollout history deployments/delivery-v010 -n backend
```

Output

```
deployment.extensions/delivery-v010
REVISION CHANGE-CAUSE
1 Initial deployment
```

You can also use the `helm list` command to view the revision history:

Bash

```
helm list
```

Output

NAME	REVISION	UPDATED	STATUS	CHART
APP VERSION	NAMESPACE			
delivery-v0.1.0	1	Sun Apr 7 00:25:30 2020	DEPLOYED	
delivery-v0.1.0	v0.1.0	backend		

## Azure DevOps Pipeline

In Azure Pipelines, pipelines are divided into *build pipelines* and *release pipelines*. The build pipeline runs the CI process and creates build artifacts. For a microservices architecture on Kubernetes, these artifacts are the container images and Helm charts that define each microservice. The release pipeline runs that CD process that deploys a microservice into a cluster.

Based on the CI flow described earlier in this article, a build pipeline might consist of the following tasks:

1. Build the test runner container.

YAML

```
- task: Docker@1
 inputs:
 azureSubscriptionEndpoint: $(AzureSubscription)
 azureContainerRegistry: $(AzureContainerRegistry)
 arguments: '--pull --target testrunner'
```

```
dockerFile: $(System.DefaultWorkingDirectory)/$(dockerFileName)
imageName: '$(imageName)-test'
```

2. Run the tests, by invoking docker run against the test runner container.

YAML

```
- task: Docker@1
 inputs:
 azureSubscriptionEndpoint: $(AzureSubscription)
 azureContainerRegistry: $(AzureContainerRegistry)
 command: 'run'
 containerName: testrunner
 volumes:
 '$(System.DefaultWorkingDirectory)/TestResults:/app/tests/TestResults'
 imageName: '$(imageName)-test'
 runInBackground: false
```

3. Publish the test results. See [Build an image](#).

YAML

```
- task: PublishTestResults@2
 inputs:
 testResultsFormat: 'VSTest'
 testResultsFiles: 'TestResults/*.trx'
 searchFolder: '$(System.DefaultWorkingDirectory)'
 publishRunAttachments: true
```

4. Build the runtime container.

YAML

```
- task: Docker@1
 inputs:
 azureSubscriptionEndpoint: $(AzureSubscription)
 azureContainerRegistry: $(AzureContainerRegistry)
 dockerFile: $(System.DefaultWorkingDirectory)/$(dockerFileName)
 includeLatestTag: false
 imageName: '$(imageName)'
```

5. Push the container image to Azure Container Registry (or other container registry).

YAML

```
- task: Docker@1
 inputs:
 azureSubscriptionEndpoint: $(AzureSubscription)
 azureContainerRegistry: $(AzureContainerRegistry)
```

```
 command: 'Push an image'
 imageName: '$(imageName)'
 includeSourceTags: false
```

## 6. Package the Helm chart.

YAML

```
- task: HelmDeploy@0
 inputs:
 command: package
 chartPath: $(chartPath)
 chartVersion: $(Build.SourceBranchName)
 arguments: '--app-version $(Build.SourceBranchName)'
```

## 7. Push the Helm package to Azure Container Registry (or other Helm repository).

YAML

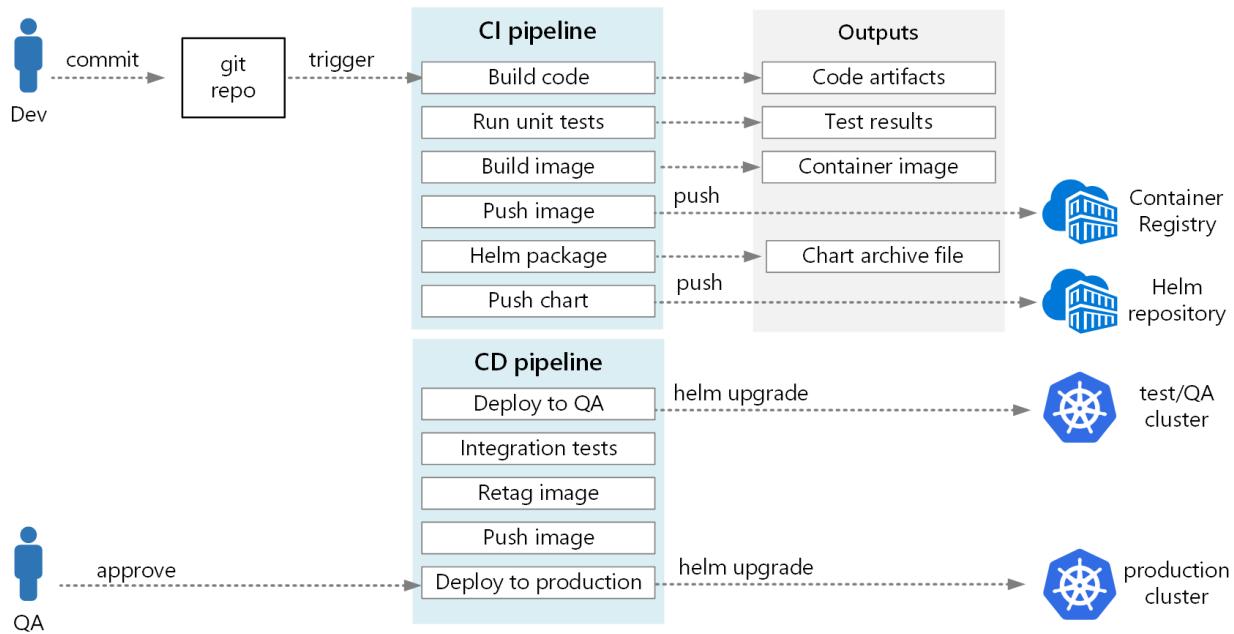
```
task: AzureCLI@1
 inputs:
 azureSubscription: $(AzureSubscription)
 scriptLocation: inlineScript
 inlineScript: |
 az acr helm push
 $(System.ArtifactsDirectory)/$(repositoryName)-$(Build.SourceBranchName)
 .tgz --name $(AzureContainerRegistry);
```

The output from the CI pipeline is a production-ready container image and an updated Helm chart for the microservice. At this point, the release pipeline can take over. There will be a unique release pipeline for each microservice. The release pipeline will be configured to have a trigger source set to the CI pipeline that published the artifact. This pipeline allows you to have independent deployments of each microservice. The release pipeline performs the following steps:

- Deploy the Helm chart to dev/QA/staging environments. The `Helm upgrade` command can be used with the `--install` flag to support the first install and subsequent upgrades.
- Wait for an approver to approve or reject the deployment.
- Retag the container image for release
- Push the release tag to the container registry.
- Deploy the Helm chart in the production cluster.

For more information about creating a release pipeline, see [Release pipelines, draft releases, and release options](#).

The following diagram shows the end-to-end CI/CD process described in this article:



## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [John Poole](#) | Senior Cloud Solutions Architect

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

- [Adopt a Git branching strategy](#)
- [What is Azure Pipelines?](#)
- [Release pipelines, draft releases, and release options](#)
- [Release deployment control using approvals](#)
- [Introduction to Container registries in Azure](#)

## Related resources

- [CI/CD for microservices](#)
- [Monitor a microservices architecture in Azure Kubernetes Service \(AKS\)](#)
- [Review a reference architecture which shows a microservices application deployed to Azure Kubernetes Service \(AKS\)](#)

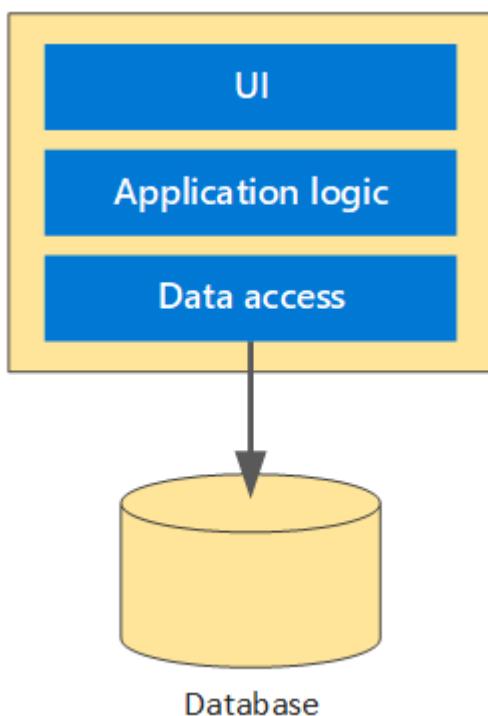
- GitOps for Azure Kubernetes Service

# Migrate a monolithic application to microservices using domain-driven design

ASP.NET

This article describes how to use domain-driven design (DDD) to migrate a monolithic application to microservices.

A monolithic application is typically an application system in which all of the relevant modules are packaged together as a single deployable unit of execution. For example, it might be a Java Web Application (WAR) running on Tomcat or an ASP.NET application running on IIS. A typical monolithic application uses a layered design, with separate layers for UI, application logic, and data access.



These systems start small but tend to grow over time to meet business needs. At some point, as new features are added, a monolithic application can begin to suffer from the following problems:

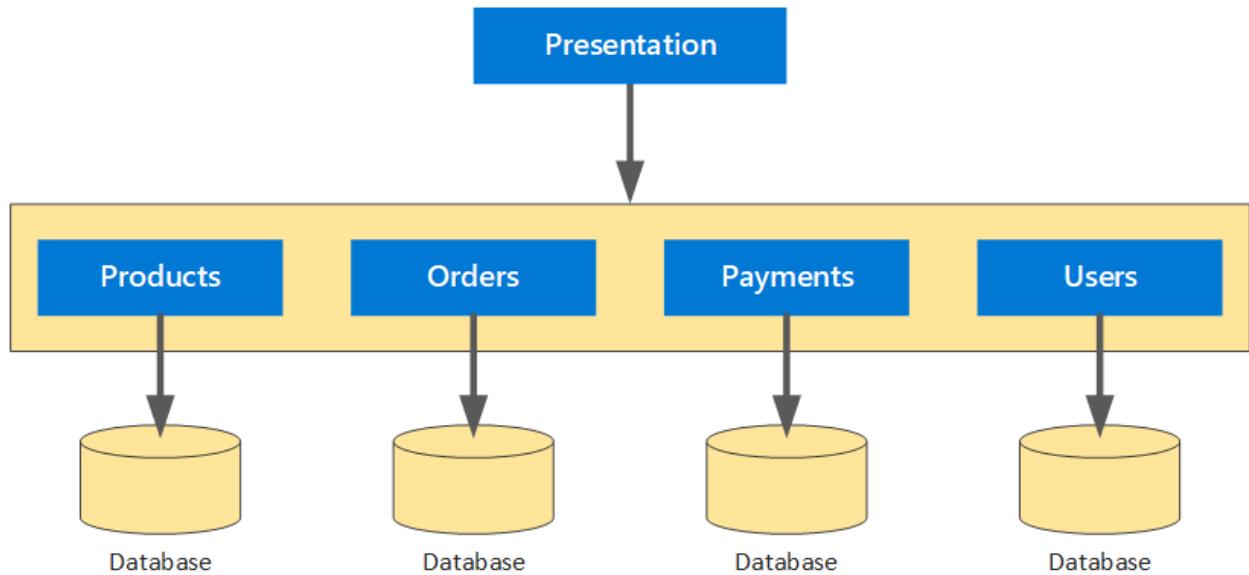
- The individual parts of the system cannot be scaled independently, because they are tightly coupled.
- It is hard to maintain the code, because of tight coupling and hidden dependencies.
- Testing becomes harder, increasing the probability of introducing vulnerabilities.

These problems can become an obstacle to future growth and stability. Teams become wary of making changes, especially if the original developers are no longer working on the project and design documents are sparse or outdated.

Despite these limitations, a monolithic design can make sense as a starting point for an application. Monoliths are often the quickest path to building a proof-of-concept or minimal viable product. In the early phases of development, monoliths tend to be:

- Easier to build, because there is a single shared code base.
- Easier to debug, because the code runs within a single process and memory space.
- Easier to reason about, because there are fewer moving parts.

As the application grows in complexity, however, these advantages can disappear. Large monoliths often become progressively harder to build, debug, and reason about. At some point, the problems outweigh the benefits. This is the point when it can make sense to migrate the application to a [microservices architecture](#). Unlike monoliths, microservices are typically decentralized, loosely coupled units of execution. The following diagram shows a typical microservices architecture:



Migrating a monolith to a microservice requires significant time and investment to avoid failures or overruns. To ensure that any migration is successful, it's good to understand both the benefits and also challenges that microservices bring. The benefits include:

- Services can evolve independently based on user needs.
- Services can scale independently to meet user demand.
- Over time, development cycles become faster as features can be released to market quicker.
- Services are isolated and are more tolerant of failure.
- A single service that fails will not bring down the entire application.

- Testing becomes more coherent and consistent, using [behavior-driven development](#).

For more information about the benefits and challenges of microservices, see [Microservices architecture style](#).

## Apply domain-driven design

Any migration strategy should allow teams to incrementally refactor the application into smaller services, while still providing continuity of service to end users. Here's the general approach:

- Stop adding functionality to the monolith.
- Split the front end from the back end.
- Decompose and decouple the monolith into a series of microservices.

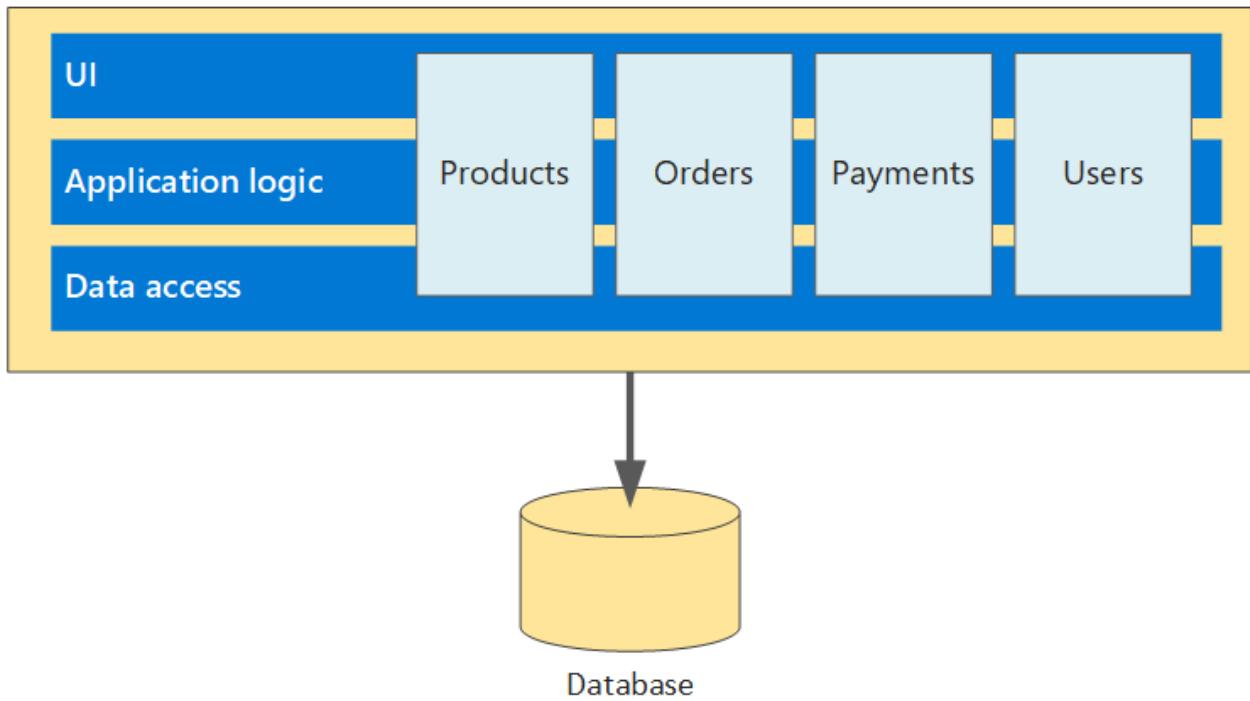
To help facilitate this decomposition, a viable software development approach is to apply the principles of domain-driven design (DDD).

Domain Driven Design (DDD) is a software development approach first introduced by [Eric Evans](#). DDD requires a good understanding of the domain for which the application will be written. The necessary domain knowledge to create the application resides within the people who understand it — the domain experts.

The DDD approach can be applied retroactively to an existing application, as a way to begin decomposing the application.

1. Start with a *ubiquitous language*, a common vocabulary that is shared between all stakeholders.
2. Identify the relevant modules in the monolithic application and apply the common vocabulary to them.
3. Define the domain models of the monolithic application. The domain model is an abstract model of the business domain.
4. Define *bounded contexts* for the models. A bounded context is the boundary within a domain where a particular domain model applies. Apply explicit boundaries with clearly defined models and responsibilities.

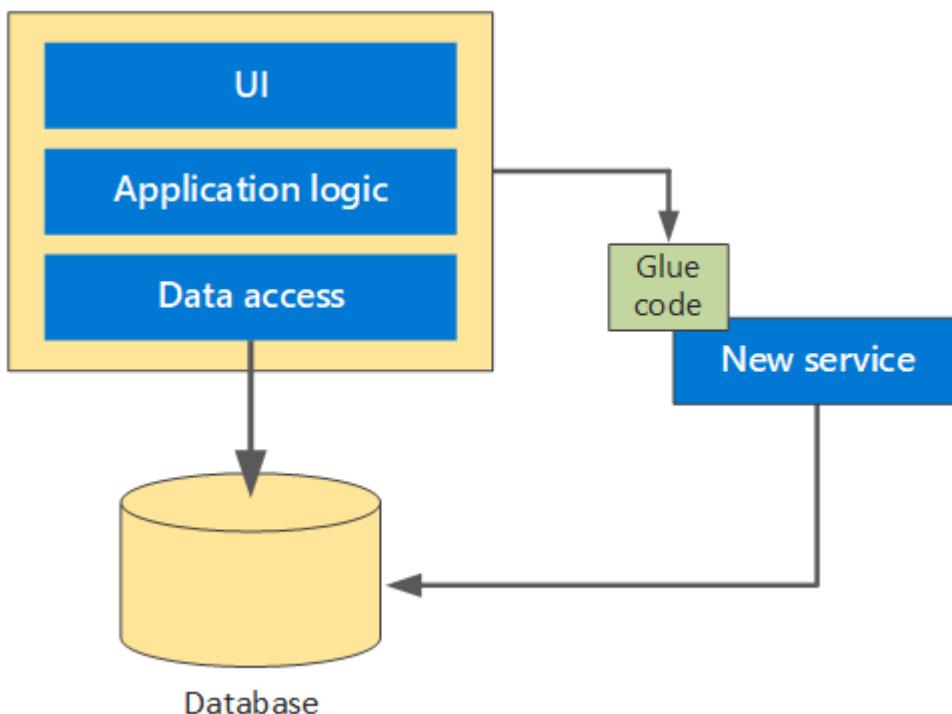
The bounded contexts identified in step 4 are candidates for refactoring into smaller microservices. The following diagram shows the existing monolith with the bounded contexts overlaid:



For more information about using a DDD approach for microservices architectures, see [Using domain analysis to model microservices](#).

## Use glue code (anti-corruption layer)

While this investigative work is carried out to inventory the monolithic application, new functionality can be added by applying the principles of DDD as separate services. "Glue code" allows the monolithic application to proxy calls to the new service to obtain new functionality.



The [glue code](#) (adapter pattern) effectively acts as an anti-corruption layer, ensuring that the new service is not polluted by data models required by the monolithic application. The glue code helps to mediate interactions between the two and ensures that only data required by the new service is passed to enable compatibility.

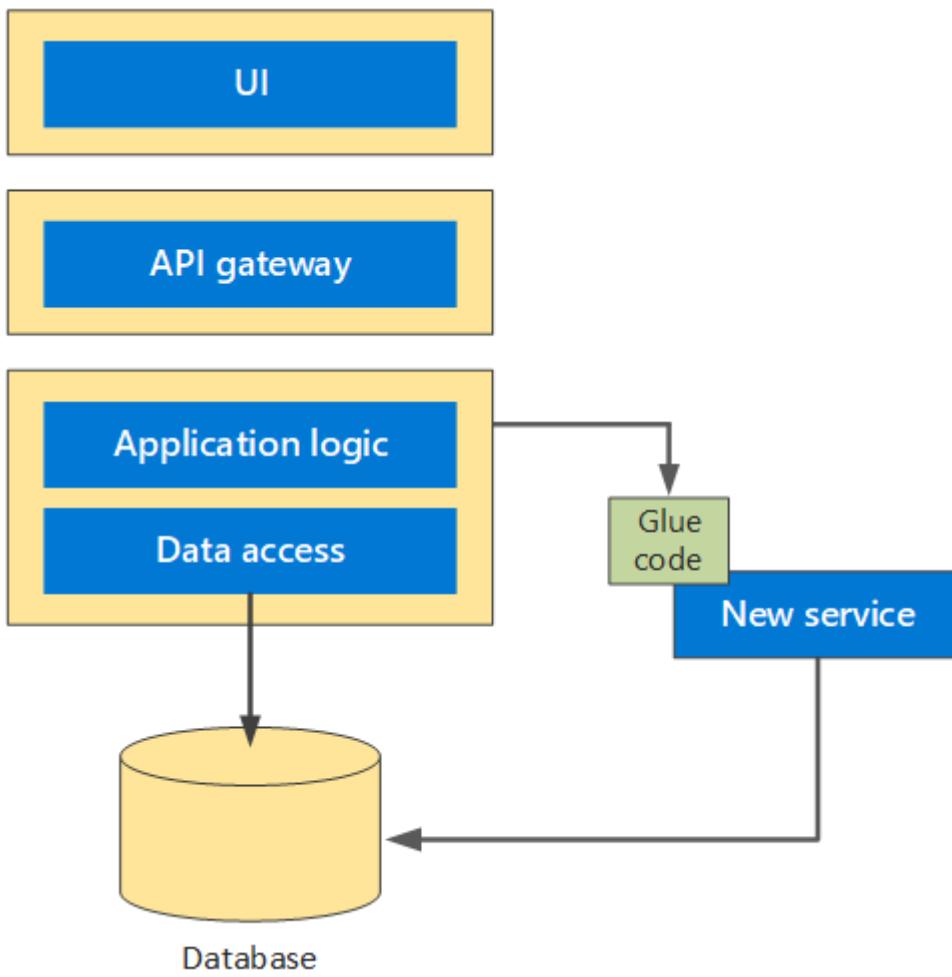
Through the process of refactoring, teams can inventory the monolithic application and identify candidates for microservices refactoring while also establishing new functionality with new services.

For more information about anti-corruption layers, see [Anti-Corruption Layer pattern](#).

## Create a presentation layer

The next step is to separate the presentation layer from the backend layer. In a traditional n-tier application, the application (business) layer tends to be the components that are core to the application and have domain logic within them. These coarse-grained APIs interact with the data access layer to retrieve persisted data from within a database. These APIs establish a natural boundary to the presentation tier, and help to decouple the presentation tier into a separate application space.

The following diagram shows the presentation layer (UI) split out from the application logic and data access layers.

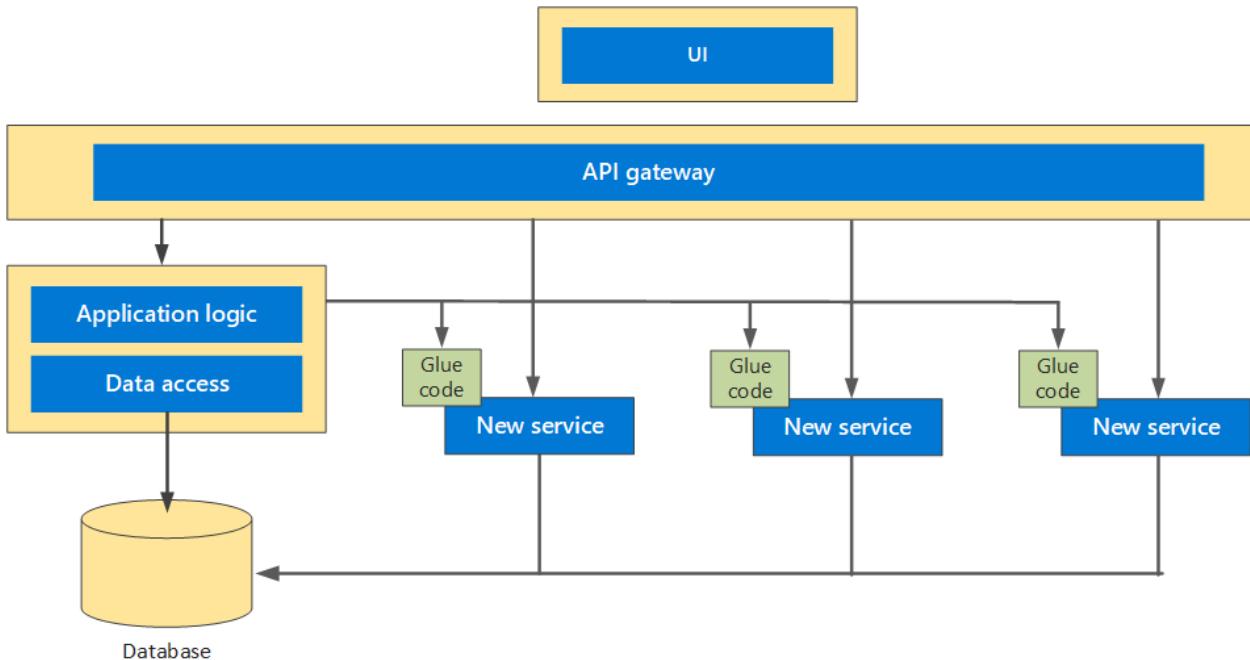


This diagram also introduces another layer, the API gateway, that sits between the presentation layer and the application logic. The API gateway is a façade layer that provides a consistent and uniform interface for the presentation layer to interact with, while allowing downstream services to evolve independently, without affecting the application. The API Gateway may use a technology such as [Azure API Management](#), and allows the application to interact in a RESTful manner.

The presentation tier can be developed in any language or framework that the team has expertise in, such as a single page application or an MVC application. These applications interact with the microservices via the gateway, using standard HTTP calls. For more information about API Gateways, see [Using API gateways in microservices](#).

## Start to retire the monolith

At this stage, the team can begin peeling away the monolithic application and slowly extract the services that have been established by their bounded contexts into their own set of microservices. The microservices can expose a RESTful interface for the application layer to interact with, through the API gateway, with glue code in place to communicate with the monolith in specific circumstances.



As you continue to peel away the monolith, eventually there will come the point when it no longer needs to exist, and the microservices have been successfully extracted from the monolith. At this point, the anti-corruption layer (glue code) can safely be removed.

This approach is an example of the [Strangler Fig pattern](#) and allows for a controlled decomposition of a monolith into a set of microservices. Over time, as existing functionality is moved into microservices, the monolith will shrink in size and complexity, to the point that it no longer exists.

## Contributors

*This article is maintained by Microsoft. It was originally written by the following contributors.*

Principal author:

- [Lavan Nallainathan](#) ↗ | Senior Cloud Solution Architect

*To see non-public LinkedIn profiles, sign in to LinkedIn.*

## Next steps

When the application has been decomposed into constituent microservices, it becomes possible to use modern orchestration tools such as [Azure DevOps](#) to manage the lifecycle of each service. For more information, see [CI/CD for microservices architectures](#).

## Related resources

- Using tactical DDD to design microservices
- Microservices architecture design
- Microservices assessment and readiness
- Design patterns for microservices

# Modernize enterprise applications with Azure Service Fabric

Azure Load Balancer

Azure Service Fabric

This article provides guidelines for moving Windows applications to an Azure compute platform without rewriting. This migration uses container support in Azure Service Fabric.

A typical approach for migrating existing workloads to the cloud is the lift-and-shift strategy. In IaaS virtual machine (VM) migrations, you provision VMs with network and storage components and deploy the existing applications onto those VMs.

Unfortunately, lift-and-shift often results in overprovisioning and overpaying for compute resources. Another approach is to move to PaaS platforms or refactor code into microservices and run in newer serverless platforms. But those options typically involve changing existing code.

Containers and container orchestration offer improvements. Containerizing an existing application enables it to run on a cluster with other applications. It provides tight control over resources, scaling, and shared monitoring and DevOps.

Optimizing and provisioning the right amount of compute resources for containerization isn't trivial. Service Fabric's orchestration allows an organization to migrate Windows and Linux applications to a runtime platform without changing code and to scale the needs of the application without overprovisioning VMs. The result is better density, better hardware use, simplified operations, and overall lower cloud-compute costs.

An enterprise can use Service Fabric as a platform to run a large set of existing Windows-based web applications with improved density, monitoring, consistency, and DevOps, all within a secure extended private network in the cloud. The principle is to use Docker and Service Fabric's containerization support that packages and hosts existing web applications on a shared cluster with shared monitoring and operations, in order to maximize cloud compute resources for the ideal performance-to-cost ratio.

This article describes the processes, capabilities, and Service Fabric features that enable containerizing in an optimal environment for a large enterprise. The guidance is scoped to web applications and Windows containers. Before reading this article, get familiar with core Windows container and Service Fabric concepts. For more information, see:

- [Create your first Service Fabric container application on Windows](#)

- [Service Fabric terminology overview](#)
- [Service Fabric best practices overview](#)

## Resources



[Sample: Modernization templates and scripts ↗](#).

The repo has these resources:

- An example Azure Resource Manager template to bring up an Azure Service Fabric cluster.
- A reverse-proxy solution for brokering web requests into the Service Fabric cluster to the destination containers.
- Sample Service Fabric application configuration and scripts that show the use of placement, resource constraints, and autoscaling.
- Sample scripts and Docker files that build and package an existing web application.

Customize the templates in this repo for your cluster. The templates implement the best practices described in this article.

## Evaluate requirements

Before containerizing existing applications, evaluate requirements. Select applications that are right for migration, choose the right developer workstation, and determine network requirements.

## Application selection

First, determine the type of applications that are best suited for a containerized platform, full virtual machines, and pure PaaS environment. The application could be a shared application that is built with Service Fabric to share Windows Server hosts across various containerized applications. Each Service Fabric host can run multiple different applications running in isolated Windows containers.

Consider creating a set of criteria to determine such applications. Here are some example criteria of containerized Windows applications in Service Fabric.

- HTTP/HTTPS web and application tiers without database dependency.
- Stateless web applications.
- Built with .NET Framework versions 3.5 and later.
- Do not have hardware dependency or access device drivers.

- Applications can run on Windows Server 2016 and later versions.
- All dependencies can be containerized, such as are most .NET assemblies, WCF, COM+.

 **Note**

Dependencies that cannot be containerized include MSMQ (Currently supported in preview releases of Windows Server Core post 1709).

- Applications can compile and build in Visual Studio.

For the web applications, databases, and other required servers (such as Active Directory) exist outside the Service Fabric cluster in IaaS VMs, PaaS, or on-premises.

## Developer workstation requirements

From an application development perspective, determine the workstation requirements.

- [Docker for Windows](#) is required for developers to containerize and test their applications prior to deployment.
- Visual Studio Docker support is required. Standardize on the latest version of [Visual Studio](#) for the best Docker compatibility.
- If workstations don't have enough hardware resources to oversee those requirements, use Azure compute resources for speed and productivity gains. An option is the Azure DevTest Labs Service. Docker for Windows, and Visual Studio 2019 require a minimum of 8 GB of memory.

## Networking requirements

Service Fabric orchestration provides a platform for hosting, deploying, scaling, and operating applications at enterprise scale. Most large enterprises that use Azure:

- Extend their corporate network with a private address space to an Azure subscription. Use either [ExpressRoute](#) or a [Site-to-Site VPN](#) to provide secure on-premises connectivity.
- Want to control inbound and outbound network traffic through third-party firewall appliances and/or [Azure Network Security Group](#) rules.
- Want tight control over the address space requirements and subnets.

Service Fabric is suitable as a containerization platform. It plugs into an existing cloud infrastructure and doesn't require open public ingress endpoints. You just need to carve

out the necessary address space for Service Fabric's IP address requirements. For details, see the [Service Fabric Networking](#) section in this article.

## Containerize existing Windows applications

After you've determined the applications that meet the selection criteria, containerize them into Docker images. The result is containerized .NET web application running in IIS where all tiers run in one container.

### ⓘ Note

You can use multiple containers; one per tier.

Here are the basic steps for containerizing an application.

1. Open the project in Visual Studio.
2. Make sure the project compiles and runs locally on the developer workstation.
3. Add a Dockerfile to the project. This Dockerfile example shows a basic .NET MVC application.

```
Dockerfile

FROM mcr.microsoft.com/dotnet/framework/aspnet:4.8
ADD PublishOutput/ /inetpub/wwwroot

add a certificate and configure SSL
SHELL ["powershell", "-command"]
RUN New-Item -ItemType Directory C:\install
ADD installwebsite.ps1 c:\install\
ADD wildcard.pfx c:\install\
Powershell script to configure SSL on the website
RUN c:\install\configurewebsite

plugin into SF healthcheck ensuring the container website is running
HEALTHCHECK --interval=30s --timeout=30s --start-period=60s --retries=3
CMD curl -f http://localhost/ || exit 1
```

4. Test locally by using Docker For Windows. The application must successfully run in a Docker container by using the Visual Studio debug experience. For more information, see [Deploy a .NET app using Docker Compose](#).
5. Build (if needed), tag, and push the tested image to a container registry, like the [Azure Container Registry](#) service. This example uses an existing Azure Container

Registry named MyAcr and Docker build/tag/push to build/deploy appA to the registry.

Bash

```
docker login myacr.azurecr.io -u myacr -p <pwd>
docker build -t appa .
docker tag appa myacr.azurecr.io/appa:1.0
docker push myacr.azurecr.io/appa:1.0
```

The image is tagged with a version number that Service Fabric references when it deploys and versions the container. Azure DevOps encapsulates and executes the manual Docker build/tag/push process. DevOps details are described in the [DevOps and CI/CD](#) section.

 **Note**

In the preceding example, the base image is "mcr.microsoft.com/dotnet/framework/aspnet:4.8" from the Microsoft Container Registry.

Here are some considerations about the base images:

- The base image could be a locked-down custom enterprise image that enforces enterprise requirements. For a shared application, isolation boundaries can be created through credentials or by using separate registry. It's recommended that enterprise-supported docker images be kept separately and stored in an isolated container registry.
- Avoid storing the registry login credentials in configuration files. Instead, use role-based access control (RBAC) and [Microsoft Entra service principals](#) with Azure Container Registry. Provide read-only access to registries depending on your enterprise requirements.

For information about running an IIS ASP.NET MVC application in a Windows container, see [Migrating ASP.NET MVC Applications to Windows Containers](#).

## Service Fabric cluster configuration for enterprise deployments

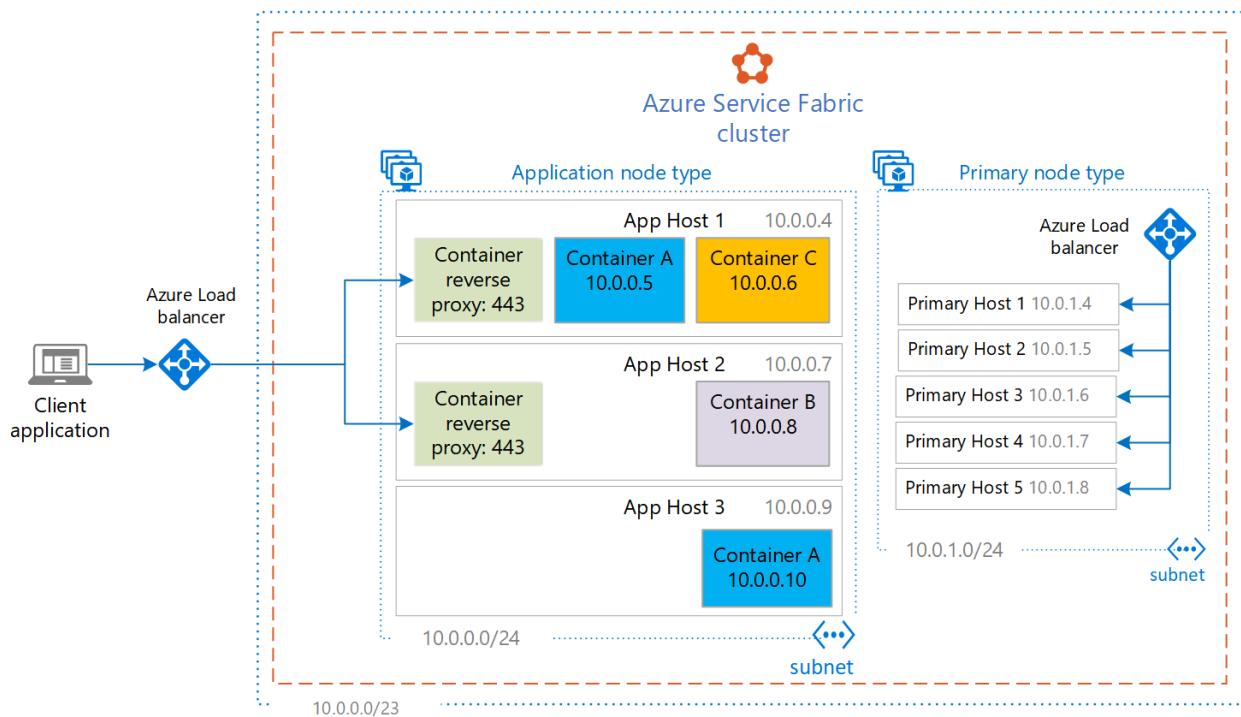
To deploy a Service Fabric cluster, start with the sample Azure Resource Manager template in this [GitHub Repo](#) and customize it to fit your requirements. You also

deploy a cluster through the Azure portal, but that option should be used for development/test provisioning.

## Service Fabric node types

A Service Fabric cluster contains one or more [node types](#). A node type typically maps to an [Azure virtual machine scale set](#) which is a set of one or more VM instances with same properties. The cluster must have at least one node type-primary node type, which runs the Service Fabric system services. Do not run your application container on the primary node type because the container can compete with the system services for resources. Consider, designing a cluster with two or more node types depending on the types of applications. For example, if you have two node types, run HTTP web and application tier containers on a non-primary node type.

This example infrastructure uses two Service Fabric node types: application and primary. You can scale in or scale out the scale set individually. It doesn't require extensive planning and testing up front to determine the correct application node type size (scale set instance count), because the actual size can grow and reduce in real time driven by monitoring and metrics.



In the preceding image, the application node type has three nodes that run three containerized applications A, B, and C. The containers can scale on demand based on CPU and memory usage. The hosts can scale out as more applications are deployed to the cluster. Azure Load Balancer has a public IP. The primary nodes are in an internal subnet. It has a load balancer that isn't publicly exposed.

## Placement constraints

Use placement constraints that target the non-primary node type to reserve the primary node type for system services.

This approach enables you to configure each scale set with a separate virtual network subnet. A unique subnet for each node type uses network security group rules for inbound/outbound access to and from the subnet and controls network flow. You can configure the primary node type with a private load balancer so that external traffic can't access Service Fabric management and application deployment. If the application node type wants to expose application endpoints publicly, configure it with a separate load balancer with security configuration. For an example template that uses Windows Service with NSG rule and multiple node types, see [7 Node, 3 node type secure Windows Service Fabric Cluster with NSG](#).

A scale set associated with a node type can reliably scale out to 100 VM instances by using a single placement group as applications are added to the cluster. The primary node type often doesn't need as many instances and can run with 5-7 nodes, depending on durability and reliability requirements.

## Service Fabric networking

Service Fabric supports with two networking modes for containerized applications; nat and Open. For large enterprise clusters that host multiple applications, use the Open mode. For more information, see [Container Networking and Constraints](#).

- **NAT**

By default, the cluster brings up containers by using a NAT-bridge mode to the host VM. The NAT bridge routes requests over a defined port to the container. With this mode, only one IP address is needed per host VM for the host's primary NIC.

To route traffic to each application container, a unique port is exposed through the load balancer. However, that port is exposed to end users. If you don't want the port exposed, provide a URL rewrite mechanism. Rewrite the application domain name with a unique application port. Traffic is routed to the load balancer that front ends the cluster. One option for the rewrite mechanism is [Azure Application Gateway](#).

Another benefit of this approach is simplistic load balancing with Azure Load Balancer. The load balancer's probe mechanism balance traffic across the VM instances that are running the application's containers.

- **Open**

The **Open** mode assigns an IP address to each running container on the host VM from the cluster's virtual network subnet. Each host is pre-allocated with a set of IP addresses. Each container on the host is assigned an IP from the virtual network range. You can configure this mode in the cluster Azure Resource Manager template during cluster creation. The example infrastructure demonstrates the **Open** mode.

Benefits of the Open mode:

- Makes connecting to application containers simple.
- Provides application traceability that is, the assigned enterprise-friendly IP is constant for the life of the container.
- Is efficient with Windows containers.

There are downsides:

- The number of IP addresses must be set aside during cluster creation and that number is fixed. For example, if 10 IP addresses are assigned to each host, the host supports up to 9 application containers; one IP is reserved for the host VM's primary NIC, the remaining addresses for each container. The number of IP addresses to configure for each host is determined based on the hardware size for the node type, and the maximum number of containers on each host. Both factors depend on application size and needs. If you need more containers in the cluster, add more application node type VM instances. That is because you can't change the IP number per instance without rebuilding the cluster.
- You need a reverse proxy to route traffic to the correct destination container. The Service Fabric DNS Service can be used by the reverse proxy to look up the application container and rewrite the HTTP request to this container. That solution is more complex to implement than the **nat** mode.

For more information, see [Service Fabric container networking modes](#).

## Service Fabric runtime and Windows versions

Choose the version of Windows Server you want to run as the infrastructure host OS. The choice depends on the testing and readiness at the time of first deployment. Supported versions are Windows Server with Containers 1607, 1709, 1803.

Also choose a version of the Service Fabric runtime to start the cluster. For a new containerization initiative, use the latest version (Service Fabric 6.4 or later).

Other considerations:

- The host OS running Service Fabric can be built by using locked down enterprise images with antivirus and malware enforcement.
- The host OS for all the Service Fabric cluster VMs can be domain joined. Domain joining isn't a requirement and can add complexity to any Azure virtual machine. However, there are benefits. For example, if an application requires Windows-integrated security to connect to domain-joined resources, then the domain service account is typically used at the process level. The account is used to execute the application instead of connection string credentials and secrets. Windows containers do not currently support direct domain joining but can be configured to use [Group Managed Storage Accounts \(gMSA\)](#). The gMSA capability is supported by Service Fabric for Windows-integrated security requirements. Each containerized application in the cluster can have its own gMSA. gMSA requires the Service Fabric host VMs that run containers to be Active Directory domain joined.

## DNS service

Service Fabric has an internal [DNS service](#) that maps a containerized application name to its location in the cluster. For the [Open](#) mode, the application's IP address is used instead. This service is enabled on the cluster. If you name each service with a DNS name, traffic is routed to the application by using a reverse proxy. For information about reverse proxy, see the [Reverse proxy for inbound traffic](#) section.

## Monitoring and diagnostics

The Service Fabric Log Analytics workspace and Service Fabric solution provide detailed information about cluster events. For information about setting it up, see [Configure Azure Monitor logs to collect cluster events](#).

- Install the [monitoring agent] for Azure Log Analytics in the Service Fabric cluster. You can then use the [container monitoring solution](#) and view the running containers in the cluster.
- Use Docker performance statistics to monitor memory and CPU use for each container.
- Install the Azure Diagnostics extension that collects diagnostic data on a deployed application. For more information, see [What is Azure Diagnostics extension](#).

## Unused container images

Container images are downloaded to each Service Fabric host and can consume space on the host disk. To free up disk space, consider [image pruning](#) to remove images that

are no longer referenced and used by running containers. Configure this option in the Host section of the cluster manifest.

## Secrets and certificates management with Key Vault

Avoid hardcoded secrets in the template. Modify the Azure Resource Manager template to add certificates (cluster certificate) and secrets (host OS admin password) from Azure Key Vault during cluster deployment. To safely store and retrieve secrets:

1. Create a Key Vault in a separate resource group in the same region as the cluster.
2. Protect the Key Vault with RBAC.
3. Load the cluster certificate and cluster password in Key Vault.
4. Reference this external Key Vault from the Azure Resource Manager template.

## Cluster capacity planning

For an enterprise production cluster, start with 5-7 primary nodes, depending on the intended size of the overall cluster. For more information, see [Service Fabric cluster capacity planning considerations](#).

For a large cluster that hosts stateless web applications, the typical hardware size for the primary node type can be Standard\_D2s\_v3. For the primary node type, a 5-node cluster with the Silver durability tier should scale to containers across 50 application VM instances. Consider doing simulation tests and add more node types as partitioning needs become apparent. Also perform tests to determine the VM size for application node type and the number of running containers on each VM instance. Typically, 10 or more containers run on each VM, but that number is dependent on these factors:

- Resource needs, such as CPU and memory use, of the containerized applications. Web applications tend to be more CPU intensive than memory intensive.
- The compute size and available resources of the chosen node VM for the application node type.
- The number of IP addresses configured per node. Each container requires an IP with Open mode. Given 15 containers on a VM, 16 IP addresses must be allocated for each VM in the application node type.
- Resource needs of the underline OS. The recommendation is to leave 25% of the resources on the VM for OS system processes.

Here are some recommendations: Choose recent Windows Server builds because older builds of Windows Server (1607) use larger container image sizes compared to newer versions (1709 and later).

Select the most appropriate VM compute size and initial application node type VM count. Standard\_D8s\_v3, Standard\_D16s\_v3, and Standard\_DSv32\_v3 are example compute sizes that are known to work well for this tier for running containerized web applications.

Use premium storage for the production cluster because container images must be read quickly from disk. Attach one data disk to each VM and move the Docker EE installation to that disk. That way there is enough log space for each running container. A standard VM with the default 127-GB OS disk fills up because of container logs. A 500 GB to 1 TB data disk is recommended per VM for the application node type scale set.

## Availability and resiliency planning

Service Fabric spreads VM instances across fault and update domains to ensure maximum availability and resilience. For an enterprise-scale production cluster, as a minimum target the Silver tier (5-node cluster) for reliability and durability when hosting containerized stateless web applications.

## Scale planning

There are two aspects to consider:

- Scale in or scale out the scale set associated with the application node type by adding or removing VM instances. VMs can be added and removed automatically through autoscaling or manually through scripts. For an enterprise cluster, do not add or remove nodes frequently. It is important that you disable nodes in Service Fabric before deleting them. Do not delete seed nodes from your cluster. Monitor the cluster VMs with alerts to trigger when VM resources exceed threshold values.
- Scale the application's container count across the scale set instances in the application node type.

For the application node type, start with the minimal required nodes to support the containerized applications and ensure high availability. Have extra nodes in the cluster. A node can be removed from the cluster for maintenance and its running containers can be temporarily moved to other nodes. The cluster can grow statically using the **Add-AzureRmServiceFabricNode** cmdlet, or dynamically by using scale set autoscale capability.

Starting with Service Fabric 6.2, application containers can autoscale individually through configuration. The configuration is based on Docker statistics for CPU and memory use on the host. Together, those capabilities can optimize the compute cost.

- Rather than overprovisioning a VM to run an application, deploy a container to the cluster, monitor it, and then scale out with additional containers, as necessary. Choosing the right size is easier because Docker statistics are used to determine the number of containers.

In the [example infrastructure](#), application A has two containers deployed across the cluster that divide load. This approach allows the application and container combination to be adjusted later for optimization.

#### Note

Docker statistics showing individual container resource utilization is sent to Log Analytics and can be analyzed in Azure Monitor.

- Service Fabric offers constant monitoring and [health checks](#) across the cluster. If a node is unhealthy, applications on that node automatically move to a healthy node and the bad node stops receiving requests. Regardless of the number of containers hosting an application, Service Fabric ensures that the application is healthy and running.

For an application that is infrequently used and can be offline, run it in the cluster with just one container instance (such as application B and C). Service Fabric makes sure that the application is up and healthy during upgrades or when the container needs to move to a new VM. Health checking can reduce cost compared to hosting that application on two redundant and overprovisioned VMs in the traditional IaaS model.

## Container networking and constraints

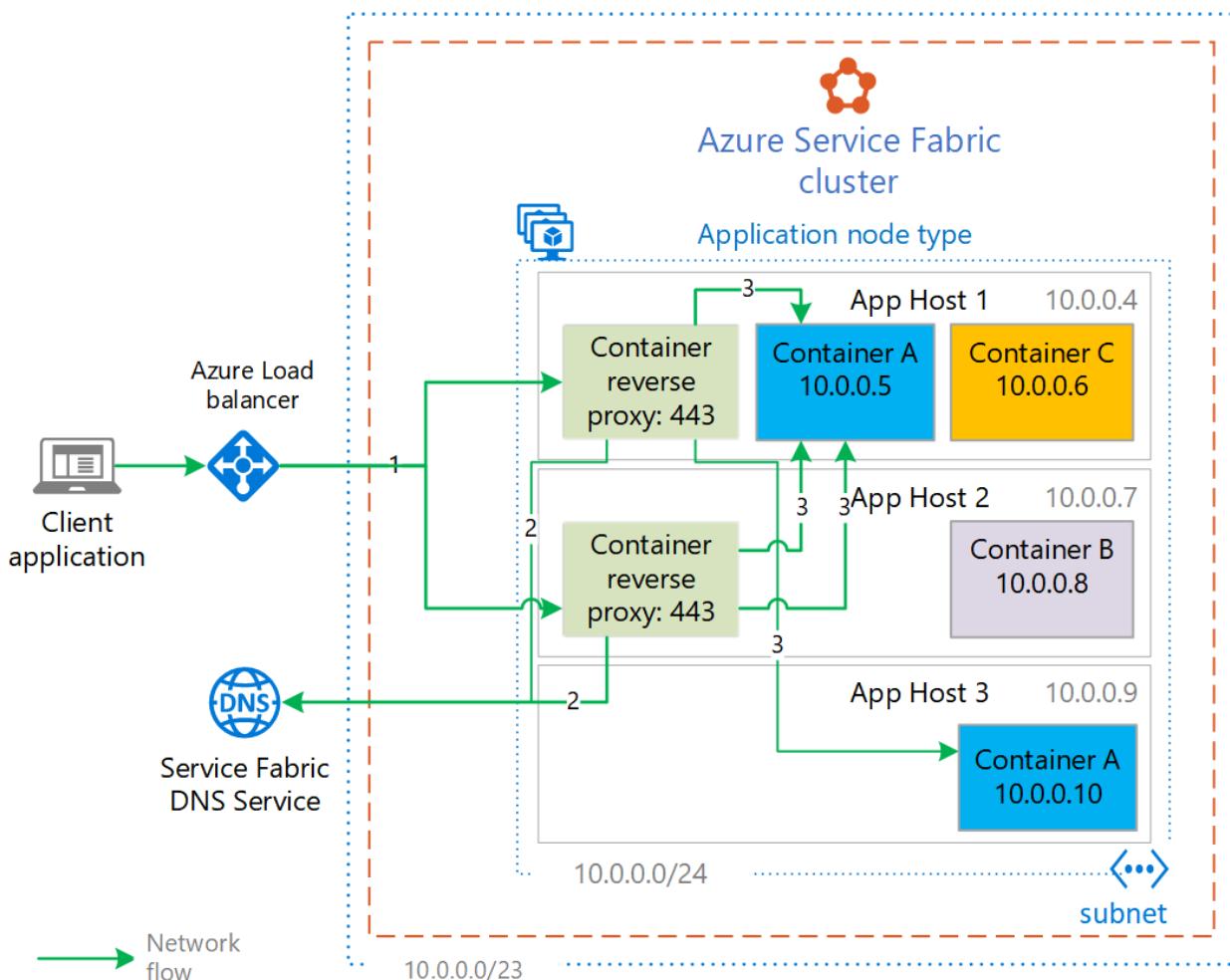
Use the [Open](#) mode for hosting containerized web applications in the cluster. After deployment, the application is immediately discoverable through the Service Fabric DNS service. The DNS service is a name-value lookup between a configured service name and the resultant IP address of a container hosting the application.

To route web requests to an application, use an ingress reverse proxy. If application containers listen on different ports (AppA port 8080, AppB on 8081), the default host NAT bridge works without issues. Azure Load Balancer probes route the traffic appropriately. However, if you want incoming traffic over SSL/443 routed to one port for all applications, use a reverse proxy to route traffic appropriately.

## Reverse proxy for inbound traffic

Service Fabric has a built-in reverse proxy but is limited in its feature set. Therefore, deploy a different reverse proxy. An option is the IIS Application Request Routing (ARR) extension for IIS hosted web applications. The ARR can be deployed to a container and configured to take inbound requests and route them to the appropriate application container. In this example, the ARR uses a NAT bridge over port 80/443, accepts all inbound web traffic, inspects the traffic, looks up the destination container using Service Fabric DNS service, and rewrites the request to the destination container. The traffic can be secured with SSL to the destination container. Follow the [IIS Application Request Routing sample](#) for building an ARR reverse proxy. For information, see [Using the Application Request Routing Module](#).

Here is the network flow for the example infrastructure.



The key aspect of the ingress reverse proxy is inspecting inbound traffic and rewriting that traffic to the destination container.

For example, application A is registered with the Service Fabric DNS service with the domain name: appA.container.myorg.com. External users access the application with <https://appA.myorg.com>. Use public or organizational DNS and register appA.myorg.com to point to the public IP for the application node type.

1. Requests for appA.myorg.com are routed to the Service Fabric cluster and handed off to the ARR container listening on port 443. Service Fabric and Azure Load Balancer set that configuration value when the ARR container is deployed.
2. When ARR gets the request, it has a condition to look for any request with the pattern='.\*', and its action rewrites the request to `https://{{C:1}}.container.{{C:2}}.{{C:3}}/{{REQUEST_URI}}`. Because the ARR is running in the cluster, the Service Fabric DNS service is invoked. The service returns the destination container IP address.
3. The request is routed to the destination container. Certificates can be used for the initial request to ARR and the rewrite to the destination container.

Here is an example ApplicationManifest.xml for Container A in the example infrastructure.

XML

```

<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest ApplicationTypeName="sfapp02Type"
 ApplicationTypeVersion="1.0.0"
 xmlns="http://schemas.microsoft.com/2011/01/fabric"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Parameters>
 <Parameter Name="appAsvc_InstanceCount" DefaultValue="2" />
 </Parameters>
 <ServiceManifestImport>
 <ServiceManifestRef ServiceManifestName="appAsvcPkg"
ServiceManifestVersion="1.0.0" />
 <ConfigOverrides />
 <Policies>
 <ContainerHostPolicies CodePackageRef="Code" Isolation="default">
 <!--SecurityOption
Value="credentialspec=file://container_gmsa1.json"-->
 <RepositoryCredentials AccountName="myacr" Password=" "
PasswordEncrypted="false"/>
 <NetworkConfig NetworkType="Open"/>
 </ContainerHostPolicies>
 <ServicePackageResourceGovernancePolicy CpuCores="1" MemoryInMB="1024"
/>
 </Policies>
 </ServiceManifestImport>
 <DefaultServices>
 <Service Name="appAsvc" ServicePackageActivationMode="ExclusiveProcess"
ServiceDnsName="appA.container.myorg.com">
 <StatelessService ServiceTypeName="appAsvcType" InstanceCount="
[appAsvc_InstanceCount]">
 <SingletonPartition />
 <PlacementConstraints>(NodeTypeNames==applicationNT)
 </PlacementConstraints>
 <StatelessService>
 </Service>
 </DefaultServices>

```

```
</DefaultServices>
</ApplicationManifest>
```

- Uses the **Open** mode for the containers.
- Registers the application domain name **appA.container.myorg.com** with the [Azure DNS service](#).
- Optionally configures the container to use an [Active Directory gMSA](#) (commented).
- Uses placement constraints to deploy the container to the application node type, named applicationNT. It instructs Service Fabric to run the container in the correct node type in the secured network subnet.
- Optionally applies [resource constraints](#). Each container is resource governed to use 1 vCPU and 1 GB of memory on the VM host. Setting a resource governance policy is recommended because Service Fabric uses the policy to distribute containers across the cluster, as opposed to the default even distribution of containers across the cluster.

Here is the example ServiceManifest.xml for the containerized application appA.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="appAsvcPkg"
 Version="1.0.0"
 xmlns="http://schemas.microsoft.com/2011/01/fabric"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <ServiceTypes>
 <StatelessServiceType ServiceTypeName="appAsvcType"
 UseImplicitHost="true" />
 </ServiceTypes>
 <CodePackage Name="Code" Version="1.0.0">
 <EntryPoint>
 <ContainerHost>
 <ImageName>myacr.azurecr.io/appa:1.0</ImageName>
 </ContainerHost>
 </EntryPoint>
 </CodePackage>
 <ConfigPackage Name="Config" Version="1.0.0" />
 <Resources>
 <Endpoints>
 <Endpoint Name="appAsvcTypeEndpoint1" UriScheme="http" Port="80"
Protocol="http" CodePackageRef="Code"/>
 <Endpoint Name="appAsvcTypeEndpoint2" UriScheme="https" Port="443"
Protocol="http" CodePackageRef="Code"/>
 </Endpoints>
 </Resources>
</ServiceManifest>
```

- The application is configured to listen on ports 80 and 443. By using the **Open** mode and reverse proxy, all applications can share the same ports.
- The application is containerized to an image named: myacr.azurecr.io/appa:1.0. Service Fabric invokes the Docker daemon to pull down the image when the application is deployed. Service Fabric handles all interactions with Docker.

The reverse proxy container uses similar manifests but isn't configured to use the **Open** mode. You can update containers by versioning the Docker image, then redeploying the versioned Service Fabric package with the **Start-ServiceFabricApplicationUpgrade** cmdlet.

For information about manifests, see [Service Fabric application and service manifests](#).

## Environmental configuration

Do not hardcode configuration values in the container image by using [environment variables](#) to pass values to a container. A DevOps pipeline can build a container image, test in a test environment, promote to staging (or pre-production), and promote to production. Do not rebuild an image for each environment.

Docker can pass environment variables directly to a container when starting one. In this example, Docker passes the `eShopTitle` variable to the `eshopweb` container:

```
Bash
```

```
docker run -p 80:80 -d --name eshoptest -e eShopTitle=SomeName eshopweb:1.0
```

In a Service Fabric cluster, Service Fabric controls Docker execution and lists environment variables in the `ServiceManifest`. Those variables are passed automatically when Service Fabric runs the container. You can override the variables in `ApplicationManifest.xml` by using the *EnvironmentOverrides* element, which can be parameterized and built from Visual Studio publish profiles for each environment.

For information about specifying environment variables in, see [How to specify environment variables for services in Service Fabric](#).

## Security considerations

Here are some articles about container security:

[Azure Service Fabric security](#)

[Service Fabric application and service security](#)

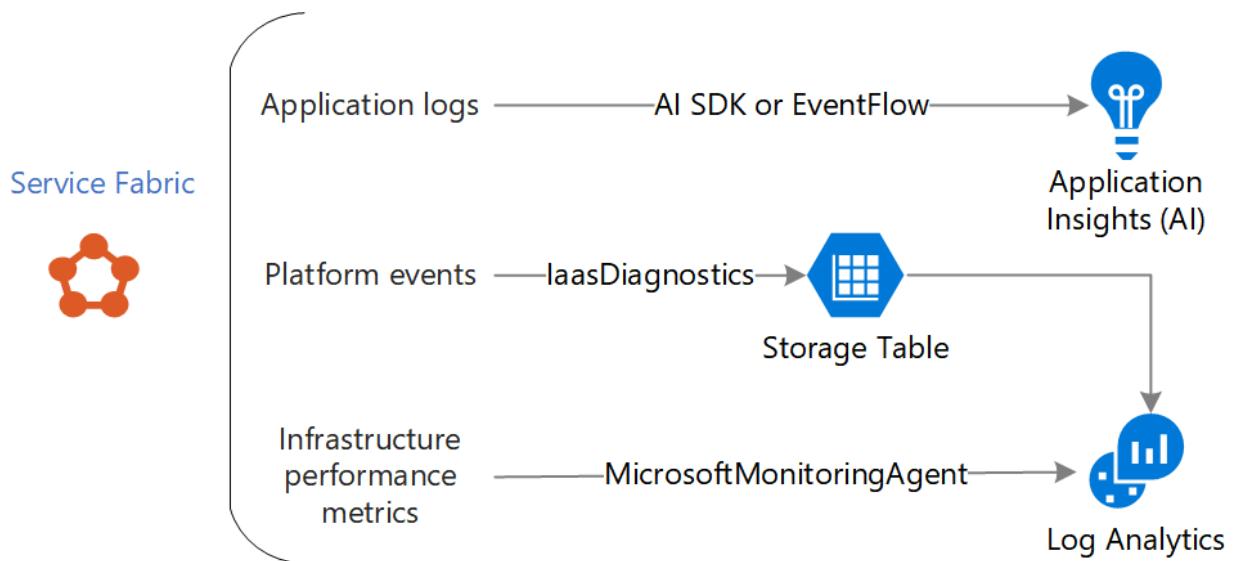
## Logging and monitoring

Monitoring and logging are critical to operational success and is achieved through integration with [Azure Monitor](#) and [Log Analytics](#).

Monitor the Service Fabric cluster and each executing containers by using the scale set extension agent for Log Analytics and its associated Container Monitoring Solution. Make sure that you configure and install the extension agent and solution during cluster creation. Docker statistics for container CPU and memory utilization are sent to Log Analytics and can be queried for proactive monitoring and alerting. Set up proactive alerts through Azure Monitor. Here are metrics that you should monitor.

- High CPU and memory utilization of a container.
- Container count hitting a per VM threshold.
- HTTP error codes and count.
- Docker Enterprise Edition (Docker EE) up or down on the host VM.
- Response time of a service.
- Host VM-based alerts on CPU, memory, disk, file consumption.

These Service Fabric virtual machine scale set extensions are installed on a typical node that results in logging.



- **ServiceFabricNode.** Links the node to a storage account (support log) for tracing support. This log is used when a ticket is opened.
- **IaaS.Diagnostics.** Collects platform events, such as ServiceFabricSystemEventTable, and stores that data in a blob storage account (app log). The account is consumed in Log Analytics.

- MicrosoftMonitoringAgent. Contains all the performance data such as Docker statistics. The data (such as ContainerInventory and ContainerLog) is sent to Log Analytics.

## Application log

If your containerized application runs in a shared cluster, you can get logs such as IIS and custom logs from the container into Log Analytics. This option is recommended because of speed, scalability, and the ability to handle large amounts of unstructured data.

Set up log rotation through Docker to keep the logs size manageable. For more information, see [Rotating Docker Logs - Keeping your overlay folder small](#).

Here are two approaches for getting application logs into Log Analytics.

- Use the existing Container Monitoring Solution installed in Log Analytics. The solution automatically sends data from the container log directories on the VM (C:\ProgramData\docker\windowsfilter\*) to the configured Log Analytics workspace. Each container creates a directory underneath the \WindowsFilter path, and the contents are streamed to Log Analytics from MicrosoftMonitoringAgent on the VM. This way you can send application logs to a shared directory(s) in the container and relay the logs by using Docker Logs to the monitored container log folders.
  1. Write a process script that runs in the container periodically and analyzes log files.
  2. Monitor the log file changes in the shared folder and write the log changes to the command window where Docker Logs can capture the information outside the container.

Each container records any output sent to the command line of a container. Access the output outside the container, which is automatically executed by Container Monitoring Solution.

Bash

```
docker logs <ContainerID>
```

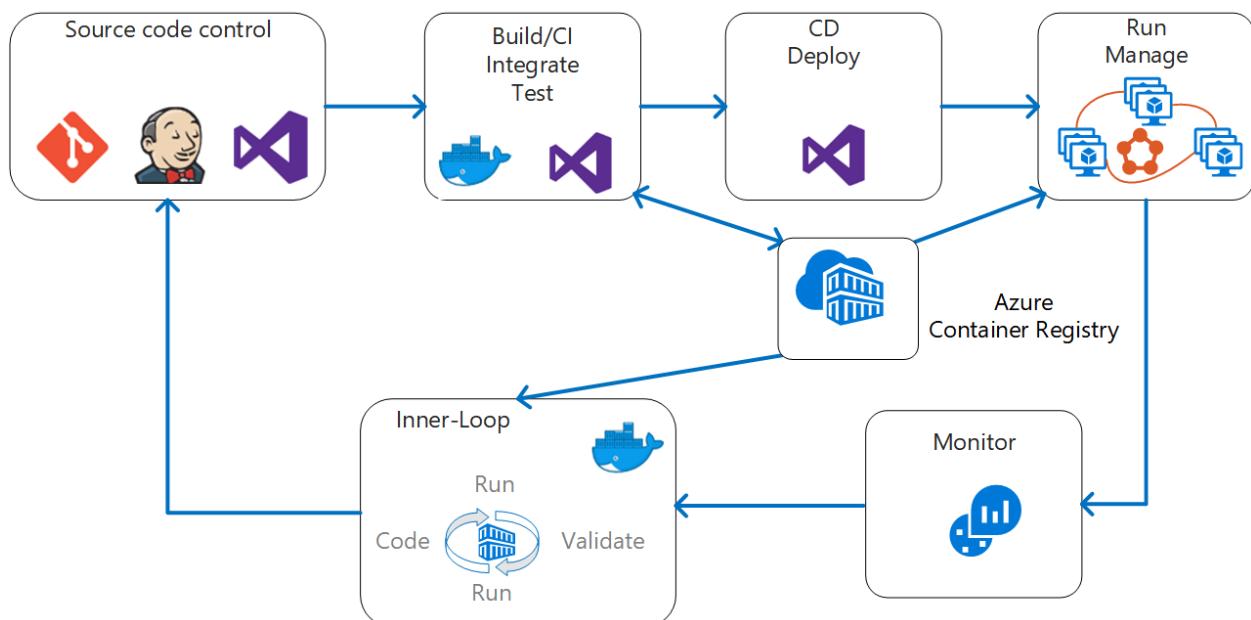
Move Docker to an attached VM data disk with enough storage to make sure the OS drive doesn't fill up with container logs.

The automatic Container Monitoring Solution sends all logs to a single Log Analytics workspace. Different containerized applications running on the same host send application logs to that shared workspace. If you need to isolate logs such that each containerized application sends the log to a separate workspace, supply a custom solution. That content is outside the scope of this article.

- Mount external storage to each running container by using a file management service such as [Azure Files](#). The container logs are sent to the external storage location and don't take up disk space on the host VM.
  - You don't need a data disk attached to each VM to hold Docker logs; move Docker Enterprise to the data disk.
  - Create a job to monitor the Azure Files location and send logs to the appropriate Log Analytics workspace for each installed application. The job doesn't need to run in the container. It just observes the Azure Files location.

## DevOps and CI/CD

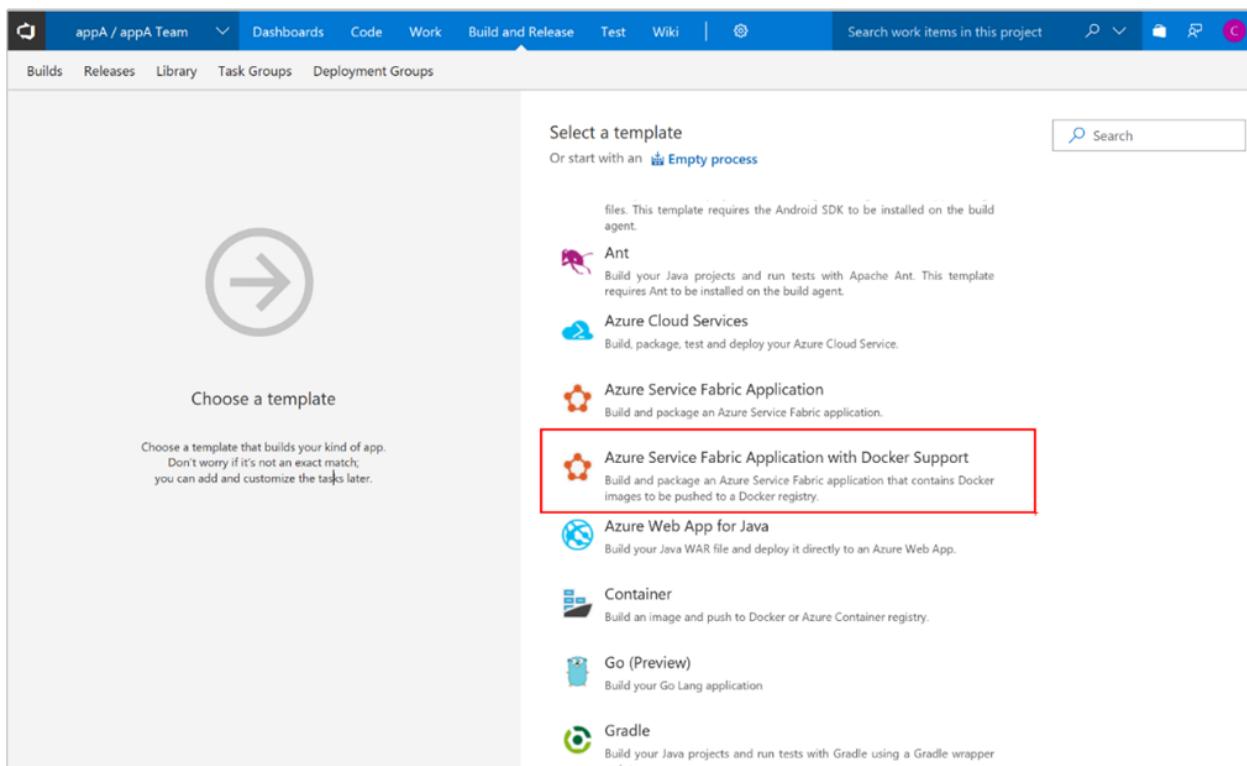
Application containerization ensures consistency. It makes sure all Service Fabric-hosted applications use the latest approved corporate image and provides an automatable image updating process that is consistent through DevOps. Azure Pipelines provides the automation process. For more information, see [Tutorial: Deploy an application with CI/CD to a Service Fabric cluster](#).



- An enterprise may want to control the base container images in a centralized registry. The preceding workflow shows one image registry. There could be multiple registries that are used to share central IT-built enterprise images with application teams. One way to centralize control is for the central IT registry to allow application teams with read-only access to the enterprise base image

repository. Application teams each have their own container registry with their Docker files and build off the central IT base image repository.

- There are various third-party image scanning tools that can plug into this process on push/pulls from the Azure Container Registry. Those solutions are available in Azure Marketplace and referenced in the Azure portal Container Registry blade. For example, Aqua and Prisma Cloud.
- After the source code is pushed to a git-based repository, set up CI/CD by creating an Azure DevOps build definition, selecting the source repository, and choosing the **Azure Service Fabric Application and Docker Support** template.



The template sets up the build process and tasks for CI/CD by building and containerizing the application, pushing the container image to a container registry (Azure Container Registry is the default), and deploying the Service Fabric application with the containerized services to the cluster. Each application code change creates a version of the code and an updated containerized image. Service Fabric's rolling upgrade feature deploys service upgrades gracefully.

The screenshot shows the Azure DevOps Build pipeline editor. The left pane displays a list of tasks in a sequence:

- Get sources (WebApplication2, master)
- Phase 1 (Run on agent):
  - NuGet restore
  - Build solution \*\*\\*.sln
  - Build solution \*\*\\*.sfproj
  - Tag images
  - Push images
  - Copy Files to: \$(build.artifactstagingdirectory)\pdbs
  - Delete files from \$(build.artifactstagingdirectory)\app...
  - Update Service Fabric Manifests (Manifest versions)
  - Update Service Fabric Manifests (Docker image settings) (selected task)
  - Copy Files to: \$(build.artifactstagingdirectory)\project...
  - Publish Artifact: drop

The right pane shows the configuration for the selected task, "Update Service Fabric Manifests (Docker image settings)":

- Version: 2.\*
- Display name: Update Service Fabric Manifests (Docker image settings)
- Update Type: Docker image settings
- Application Package: sfWebApplication2/sfWebApplication2.sfproj
- Image Digests Path: \$(build.artifactstagingdirectory)\imageDigests.txt
- Control Options
- Output Variables

Here is an example of a build starting the full DevOps process on an Azure-provided hosted build agent. Some enterprises may require the build agents to run internally within their private Azure virtual network corporate network. Set up a Windows build agent VM and instruct Azure DevOps to use the private VM for building and deploying code. For information about using custom build agents, see [Self-hosted Windows agents](#).

The screenshot shows the Azure DevOps interface for a build pipeline. The top navigation bar includes 'WebApplication2 / Web...', 'Dashboards', 'Code', 'Work', 'Build and Release', '...', and a search bar 'Search work items in this project'. Below the navigation is a secondary navigation bar with 'Builds', 'Releases', 'Library', 'Task Groups', and 'Deployment Groups'. The main content area shows a build summary for 'Build 201805011' under 'Phase 1'. The 'Job' section lists several steps: 'Initialize Agent', 'Initialize Job', 'Get Sources', 'NuGet restore', 'Build solution \*\*\\*.sln', 'Build solution \*\*\\*.sfproj', 'Tag images', 'Push images', 'Copy File to: \${build.artifactst...}', 'Delete files from \${build.artifac...}', 'Update Service Fabric Manifest...', 'Update Service Fabric Manifest...', 'Copy files to: \${build.artifactst...}', 'Publish Artifact: drop', and 'Post Job Cleanup'. The status for most steps is 'Succeeded'. The 'Build Started' section shows the build logs, which include NuGet restore, MSBuild tasks, and the start of the build solution. The logs end with a note about building the project in parallel.

```

Build Started
Job 1
Running for 69 seconds (Hosted Agent)

Console Timeline Code coverage* Tests

Adding package 'Microsoft.Net.Compilers.2.4.0' to folder 'D:\a\1\s\packages'
Added package 'Microsoft.Net.Compilers.2.4.0' to folder 'D:\a\1\s\packages'
Added package 'Microsoft.Net.Compilers.2.4.0' to folder 'D:\a\1\s\packages' from source 'https://api.nuget.org/v3/index.json'
NuGet Config files used:
 D:\a\1\NuGet\tempNuGet_70.config
Feeds used:
 C:\Users\vsadmin\NuGet\tempNuGet_70.config
 https://api.nuget.org/v3/index.json
Installed:
 21 package(s) to packages.config projects

Finishing: NuGet restore

Starting: Build solution ***.sln

Task : Visual Studio Build
Description : Build with MSBuild and set the Visual Studio version properly
Version : 1.126.0
Author : Microsoft Corporation
Help : [More Information](https://go.microsoft.com/fwlink/?LinkId=G13727)
=====
:D:\a\1\tasks\VSBUILD:1a9a2d3-a98a-4caa-96ab-affca411ecda1.126.0.ps modules\VSBUILDHelpers\vswhere.exe" -version [15.0,16.0) -latest -format json
"C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\MSBuild\15.0\bin\msbuild.exe" "D:\a\1\s\WebApplication2.sln" /nologo
/nr:raise /d1:centralLogger,"D:\a\1\tasks\VSBUILD:1a9a2d3-a98a-4caa-96ab-affca411ecda1.126.0
\ps_modules\VSBUILDHelpers\Microsoft.TeamFoundation.DistributedTask.MSBuild.Logger.dll";"RootDetailId=6633c9fb-8161-42b7-80d6-122e04fc4d47\;SolutionDir=D:\a\1\s\ForwardingLogger,"D:\a\1\tasks\VSBUILD:1a9a2d3-a98a-4caa-96ab-affca411ecda1.126.0
\ps_modules\VSBUILDHelpers\Microsoft.TeamFoundation.DistributedTask.MSBuild.Logger.dll" /p:Deterministic=true /p:PathMap=D:\a\1\c\:/p:platform=x64" /p:configuration="Release" /p:VisualStudioVersion="15.0" /p:_MSDeployUserAgent="VSTS_4167424c-aabb-4778-9760-09f453f79b3d_build_25_70"
Building the project in this solution one at a time. To enable parallel build, please add the "/m" switch.
Build started at: 5/28/2018 1:02:12 PM.
Project "D:\a\1\s\WebApplication2.sln" on node 1 (default targets).
ValidateSolutionConfiguration:
 Building solution configuration "Release|x64".
Project "D:\a\1\s\WebApplication2.sln" (1) is building "D:\a\1\s\WebApplication2\WebApplication2.csproj" (2) on node 1 (default targets).
.
PrepareForBuild:
 Creating directory "bin".
 Creating directory "obj\Release".
CoreCompile:
 D:\a\1\s\packages\Microsoft.Net.Compilers.2.4.0\build\..\tools\csc.exe /noconfig /nowarn:1701,1702 /nostdlib+ /errorreport:prompt /warn:4 /define:TRACE /highPriorityHeap /rcfrcncc:0:\a\1\s\packages\Antlr.3.4.1.0004\lib\Antlr.Runtime.dll /rcfrcncc:0:\a\1\s\packages\Microsoft.ApplicationInsights.Agent.Intercept.2.0.6\lib\net45\Microsoft.AI.Agent.Intercept.dll /reference:D:\a\1\s\packages\Microsoft.ApplicationInsights.DependencyCollector.2.0.6\lib\net45\Microsoft.AI.DependencyCollector.dll /reference:D:\a\1\s\packages\Microsoft.ApplicationInsights.PerfCounterCollector.2.0.6\lib\net45\Microsoft.AI.PerfCounterCollector.dll /reference:D:\a\1\s\packages\Microsoft.ApplicationInsights.WindowsServer.2.2.0\lib\net45\Microsoft.AI.WindowsServer.dll /reference:D:\a\1\s\packages\Microsoft.ApplicationInsights.WindowsServer.2.2.0\lib\net45\Microsoft.AI.WindowsServer.dll /reference:D:\a\1\s\packages\...

```

## Conclusion

Here is the summary of best practices:

- Before containerizing existing applications, selecting applications that are suitable for this migration, choose the right developer workstation, and determine network requirements.
- Do not run your application container on the primary node type. Instead, configure the cluster with two or more node types and run the application tier containers on a non-primary node type. Use placement constraints that target the non-primary node type to reserve the primary node type for system services.
- Use the **Open** networking mode.
- Use an ingress reverse proxy, such as the IIS Application Request Routing. The reverse proxy inspects inbound traffic and rewrites the traffic to the destination container.
- Do not hardcode configuration values in the container image and use environment variables to pass values to a container.
- Monitor the application, platform events, and infrastructure metrics by using IaaS Diagnostics and Microsoft Monitoring Agent extensions. View the logs in Application Insights and Log Analytics.

- Use the latest approved corporate image and provide an automatable image updating process that is consistent through DevOps.

## Next steps

Get the latest version of the tools you need for containerizing, such as [Visual Studio](#) and [Docker for Windows](#).

Customize these templates to meet your requirements. [Sample: Modernization templates and scripts](#).

## Related resources

- [Migration architecture design](#)
- [Build migration plan with Azure Migrate](#)
- [Modernize .NET applications](#)

# Migrate an Azure Cloud Services application to Azure Service Fabric

Azure Service Fabric



[Sample code](#) ↗

This article describes migrating an application from Azure Cloud Services to Azure Service Fabric. It focuses on architectural decisions and recommended practices.

For this project, we started with a Cloud Services application called Surveys and ported it to Service Fabric. The goal was to migrate the application with as few changes as possible. Later in the article, we show how to optimize the application for Service Fabric.

Before reading this article, it will be useful to understand the basics of Service Fabric. See [Overview of Azure Service Fabric](#)

## About the Surveys application

A fictional company named Tailspin created an application called the Surveys app that allows customers to create surveys. After a customer signs up for the application, users can create and publish surveys, and collect the results for analysis. The application includes a public website where people can take a survey. Read more about the original Tailspin scenario [here](#).

Now Tailspin wants to move the Surveys application to a microservices architecture, using Service Fabric running on Azure. Because the application is already deployed as a Cloud Services application, Tailspin adopts a multi-phase approach:

1. Port the cloud services to Service Fabric, while minimizing changes to the application.
2. Optimize the application for Service Fabric, by moving to a microservices architecture.

In a real-world project, it's likely that both stages would overlap. While porting to Service Fabric, you would also start to rearchitect the application into micro-services. Later you might refine the architecture further, perhaps dividing coarse-grained services into smaller services.

The application code is available on [GitHub](#). This repo contains both the Cloud Services application and the Service Fabric version.

## Why Service Fabric?

Service Fabric is a good fit for this project, because most of the features needed in a distributed system are built into Service Fabric, including:

- **Cluster management.** Service Fabric automatically handles node failover, health monitoring, and other cluster management functions.
- **Horizontal scaling.** When you add nodes to a Service Fabric cluster, the application automatically scales, as services are distributed across the new nodes.
- **Service discovery.** Service Fabric provides a discovery service that can resolve the endpoint for a named service.
- **Stateless and stateful services.** Stateful services use [reliable collections](#), which can take the place of a cache or queue, and can be partitioned.
- **Application lifecycle management.** Services can be upgraded independently and without application downtime.
- **Service orchestration** across a cluster of machines.
- **Higher density** for optimizing resource consumption. A single node can host multiple services.

Service Fabric is used by various Microsoft services, including Azure SQL Database, Azure Cosmos DB, Azure Event Hubs, and others, making it a proven platform for building distributed cloud applications.

## Comparing Cloud Services with Service Fabric

The following table summarizes some of the important differences between Cloud Services and Service Fabric applications. For a more in-depth discussion, see [Learn about the differences between Cloud Services and Service Fabric before migrating applications](#).

[] [Expand table](#)

Area	Cloud Services	Service Fabric
Application composition	Roles	Services

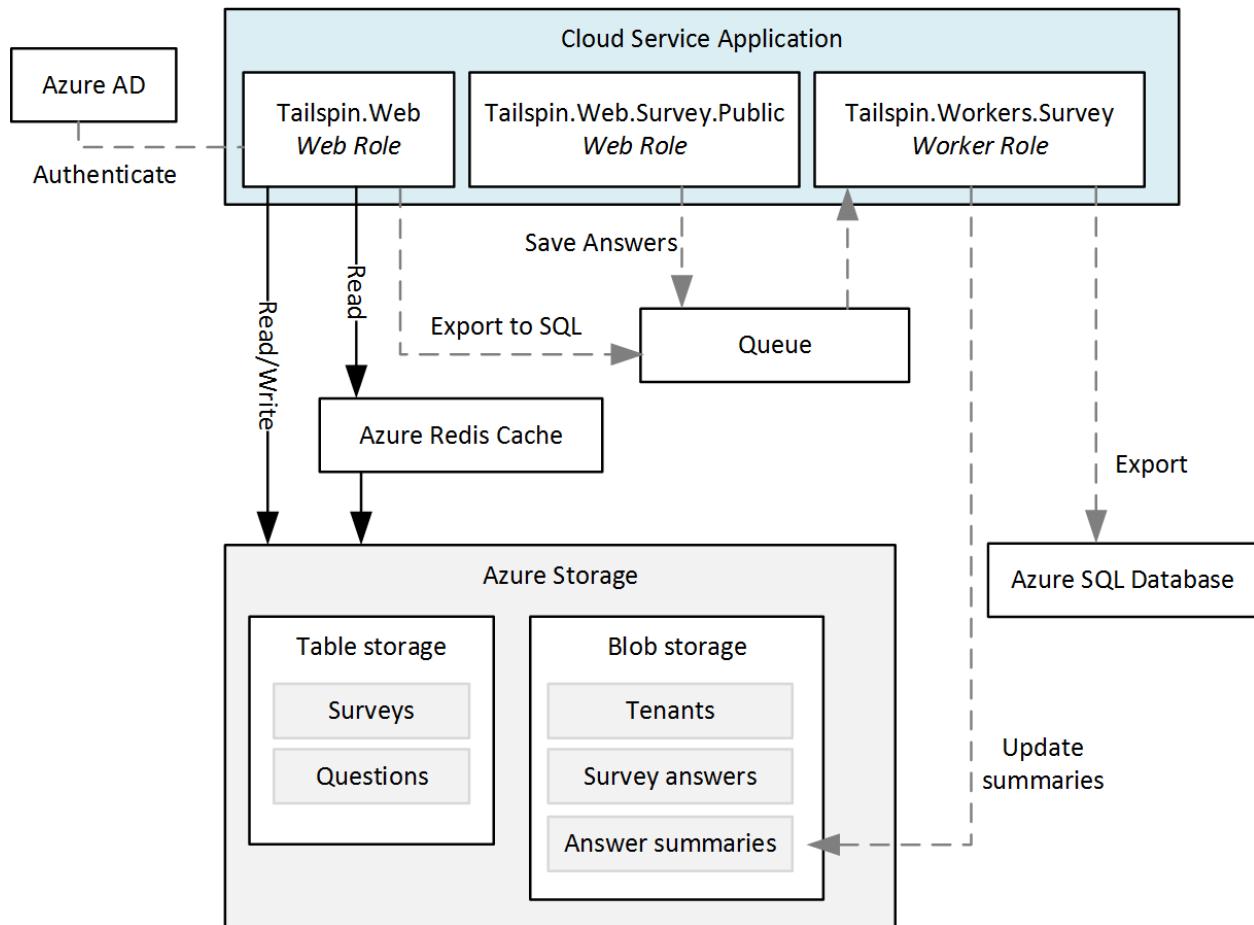
Area	Cloud Services	Service Fabric
Density	One role instance per VM	Multiple services in a single node
Minimum number of nodes	2 per role	5 per cluster, for production deployments
State management	Stateless	Stateless or stateful*
Hosting	Azure	Cloud or on-premises
Web hosting	IIS**	Self-hosting
Deployment model	<a href="#">Classic deployment model</a>	<a href="#">Resource Manager</a>
Packaging	Cloud service package files (.cspkg)	Application and service packages
Application update	VIP swap or rolling update	Rolling update
Autoscaling	<a href="#">Built-in service</a>	Virtual machine scale sets for auto scale out
Debugging	Local emulator	Local cluster

\* Stateful services use [reliable collections](#) to store state across replicas, so that all reads are local to the nodes in the cluster. Writes are replicated across nodes for reliability. Stateless services can have external state, using a database or other external storage.

\*\* Worker roles can also self-host ASP.NET Web API using OWIN.

## The Surveys application on Cloud Services

The following diagram shows the architecture of the Surveys application running on Cloud Services.



The application consists of two web roles and a worker role.

- The **Tailspin.Web** web role hosts an ASP.NET website that Tailspin customers use to create and manage surveys. Customers also use this website to sign up for the application and manage their subscriptions. Finally, Tailspin administrators can use it to see the list of tenants and manage tenant data.
- The **Tailspin.Web.Survey.Public** web role hosts an ASP.NET website where people can take the surveys that Tailspin customers publish.
- The **Tailspin.Workers.Survey** worker role does background processing. The web roles put work items onto a queue, and the worker role processes the items. Two background tasks are defined: Exporting survey answers to Azure SQL Database, and calculating statistics for survey answers.

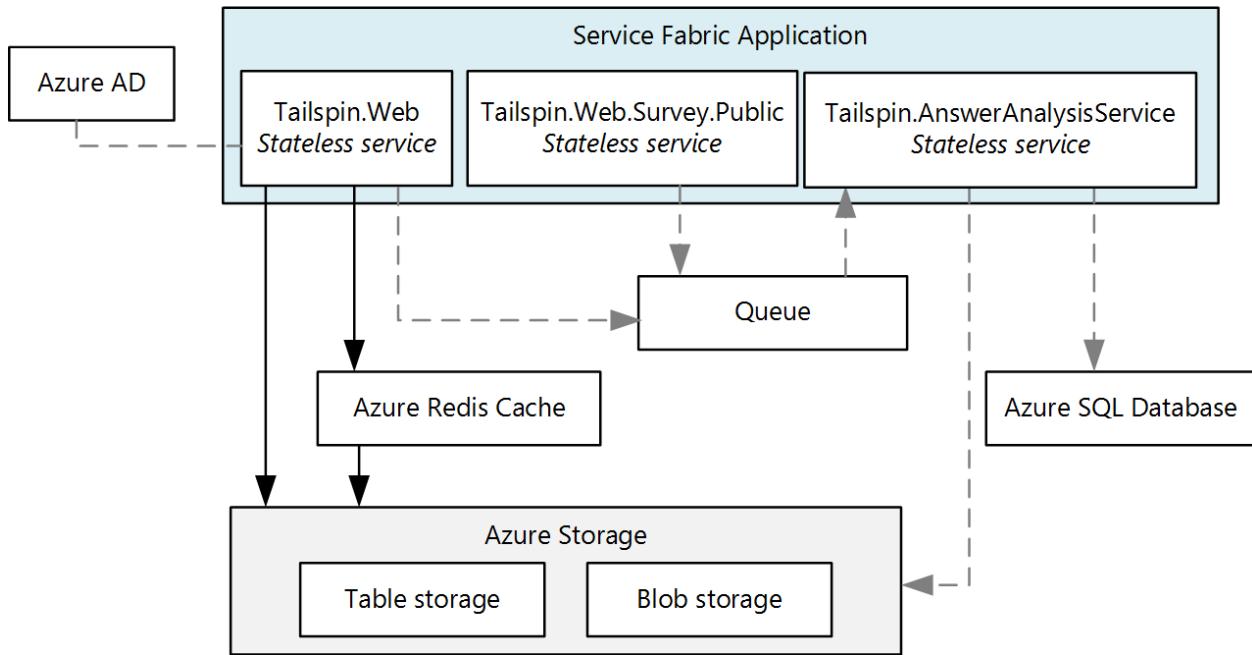
In addition to Cloud Services, the Surveys application uses some other Azure services:

- **Azure Storage** to store surveys, surveys answers, and tenant information.
- **Azure Cache for Redis** to cache some of the data that is stored in Azure Storage, for faster read access.
- **Microsoft Entra ID** (Microsoft Entra ID) to authenticate customers and Tailspin administrators.

- Azure SQL Database to store the survey answers for analysis.

## Moving to Service Fabric

As mentioned, the goal of this phase was migrating to Service Fabric with the minimum necessary changes. To that end, we created stateless services corresponding to each cloud service role in the original application:



Intentionally, this architecture is similar to the original application. However, the diagram hides some important differences. In the rest of this article, we'll explore those differences.

## Converting the cloud service roles to services

For the initial migration, Tailspin followed the steps outlined in [Guide to converting Web and Worker Roles to Service Fabric stateless services](#).

### Creating the web front-end services

In Service Fabric, a service runs inside a process created by the Service Fabric runtime. For a web front end, that means the service is not running inside IIS. Instead, the service must host a web server. This approach is called *self-hosting*, because the code that runs inside the process acts as the web server host.

The original Surveys application uses ASP.NET MVC. Because ASP.NET MVC cannot be self-hosted in Service Fabric, Tailspin considered the following migration options:

- Port the web roles to ASP.NET Core, which can be self-hosted.
- Convert the web site into a single-page application (SPA) that calls a web API implemented using ASP.NET Web API. This would have required a complete redesign of the web front end.
- Keep the existing ASP.NET MVC code and deploy IIS in a Windows Server container to Service Fabric. This approach would require little or no code change.

The first option, porting to ASP.NET Core, allowed Tailspin to take advantage of the latest features in ASP.NET Core. To do the conversion, Tailspin followed the steps described in [Migrating From ASP.NET MVC to ASP.NET Core MVC](#).

### ⓘ Note

When using ASP.NET Core with Kestrel, you should place a reverse proxy in front of Kestrel to handle traffic from the Internet, for security reasons. For more information, see [Kestrel web server implementation in ASP.NET Core](#). The section [Deploying the application](#) describes a recommended Azure deployment.

## HTTP listeners

In Cloud Services, a web or worker role exposes an HTTP endpoint by declaring it in the [service definition file](#). A web role must have at least one endpoint.

### XML

```
<!-- Cloud service endpoint -->
<Endpoints>
 <InputEndpoint name="HttpIn" protocol="http" port="80" />
</Endpoints>
```

Similarly, Service Fabric endpoints are declared in a service manifest:

### XML

```
<!-- Service Fabric endpoint -->
<Endpoints>
 <Endpoint Protocol="http" Name="ServiceEndpoint" Type="Input"
 Port="8002" />
</Endpoints>
```

Unlike a cloud service role, Service Fabric services can be colocated within the same node. Therefore, every service must listen on a distinct port. Later in this article, we'll

discuss how client requests on port 80 or port 443 get routed to the correct port for the service.

A service must explicitly create listeners for each endpoint. The reason is that Service Fabric is agnostic about communication stacks. For more information, see [Build a web service front end for your application using ASP.NET Core](#).

## Packaging and configuration

A cloud service contains the following configuration and package files:

[] [Expand table](#)

File	Description
Service definition (.csdef)	Settings used by Azure to configure the cloud service. Defines the roles, endpoints, startup tasks, and the names of configuration settings.
Service configuration (.cscfg)	Per-deployment settings, including the number of role instances, endpoint port numbers, and the values of configuration settings.
Service package (.cspkg)	Contains the application code and configurations, and the service definition file.

There is one .csdef file for the entire application. You can have multiple .cscfg files for different environments, such as local, test, or production. When the service is running, you can update the .cscfg but not the .csdef. For more information, see [What is the Cloud Service model and how do I package it?](#)

Service Fabric has a similar division between a service *definition* and service *settings*, but the structure is more granular. To understand Service Fabric's configuration model, it helps to understand how a Service Fabric application is packaged. Here is the structure:

text
<pre>Application package   - Service packages   - Code package   - Configuration package   - Data package (optional)</pre>

The application package is what you deploy. It contains one or more service packages. A service package contains code, configuration, and data packages. The code package contains the binaries for the services, and the configuration package contains configuration settings. This model allows you to upgrade individual services without redeploying the entire application. It also lets you update just the configuration settings, without redeploying the code or restarting the service.

A Service Fabric application contains the following configuration files:

[+] [Expand table](#)

File	Location	Description
ApplicationManifest.xml	Application package	Defines the services that compose the application.
ServiceManifest.xml	Service package	Describes one or more services.
Settings.xml	Configuration package	Contains configuration settings for the services defined in the service package.

For more information, see [Model an application in Service Fabric](#).

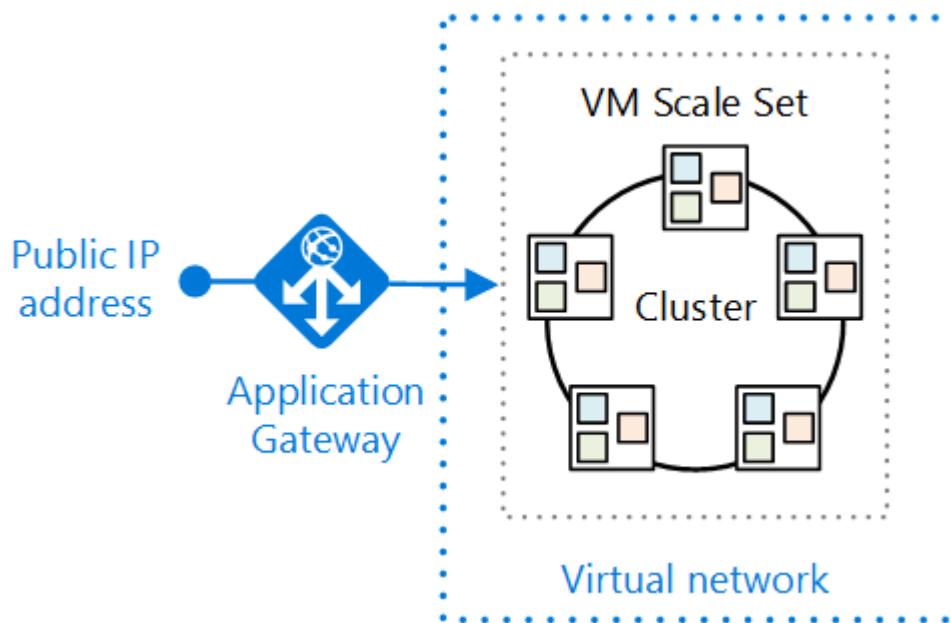
To support different configuration settings for multiple environments, use the following approach, described in [Manage application parameters for multiple environments](#):

1. Define the setting in the Setting.xml file for the service.
2. In the application manifest, define an override for the setting.
3. Put environment-specific settings into application parameter files.

## Deploying the application

Whereas Azure Cloud Services is a managed service, Service Fabric is a runtime. You can create Service Fabric clusters in many environments, including Azure and on premises.

The following diagram shows a recommended deployment for Azure:



The Service Fabric cluster is deployed to a [virtual machine scale set](#). Scale sets are an Azure Compute resource that can be used to deploy and manage a set of identical VMs.

As mentioned, it's recommended to place the Kestrel web server behind a reverse proxy for security reasons. This diagram shows [Azure Application Gateway](#), which is an Azure service that offers various layer 7 load-balancing capabilities. It acts as a reverse-proxy service, terminating the client connection and forwarding requests to back-end endpoints. You might use a different reverse proxy solution, such as nginx.

## Layer 7 routing

In the [original Surveys application](#), one web role listened on port 80, and the other web role listened on port 443.

[Expand table](#)

Public site	Survey management site
<a href="http://tailspin.cloudapp.net">http://tailspin.cloudapp.net</a>	<a href="https://tailspin.cloudapp.net">https://tailspin.cloudapp.net</a>

Another option is to use layer 7 routing. In this approach, different URL paths get routed to different port numbers on the back end. For example, the public site might use URL paths starting with `/public/`.

Options for layer 7 routing include:

- Use Application Gateway.
- Use a network virtual appliance (NVA), such as nginx.

- Write a custom gateway as a stateless service.

Consider this approach if you have two or more services with public HTTP endpoints, but want them to appear as one site with a single domain name.

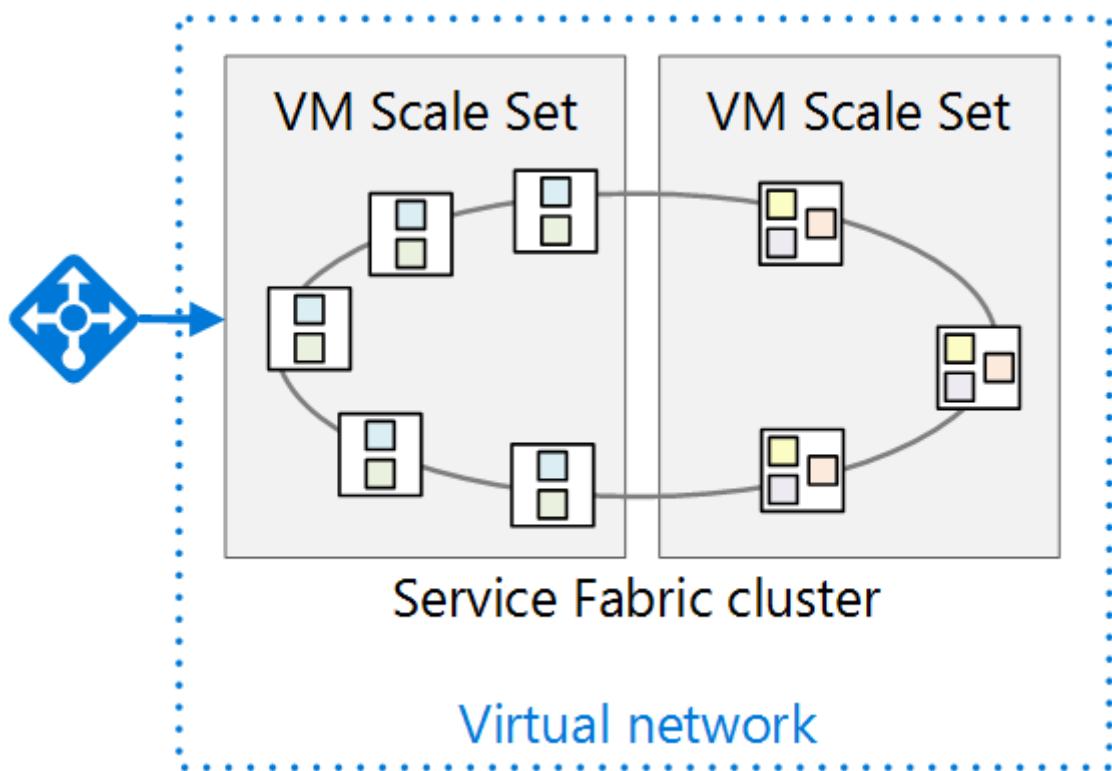
One approach that we *don't* recommend is allowing external clients to send requests through the Service Fabric [reverse proxy](#). Although this is possible, the reverse proxy is intended for service-to-service communication. Opening it to external clients exposes *any* service running in the cluster that has an HTTP endpoint.

## Node types and placement constraints

In the deployment shown above, all the services run on all the nodes. However, you can also group services, so that certain services run only on particular nodes within the cluster. Reasons to use this approach include:

- Run some services on different VM types. For example, some services might be compute-intensive or require GPUs. You can have a mix of VM types in your Service Fabric cluster.
- Isolate front-end services from back-end services, for security reasons. All the front-end services will run on one set of nodes, and the back-end services will run on different nodes in the same cluster.
- Different scale requirements. Some services might need to run on more nodes than other services. For example, if you define front-end nodes and back-end nodes, each set can be scaled independently.

The following diagram shows a cluster that separates front-end and back-end services:



To implement this approach:

1. When you create the cluster, define two or more node types.
2. For each service, use [placement constraints](#) to assign the service to a node type.

When you deploy to Azure, each node type is deployed to a separate virtual machine scale set. The Service Fabric cluster spans all node types. For more information, see [The relationship between Service Fabric node types and virtual machine scale sets](#).

If a cluster has multiple node types, one node type is designated as the *primary* node type. Service Fabric runtime services, such as the Cluster Management Service, run on the primary node type. Provision at least 5 nodes for the primary node type in a production environment. The other node type should have at least 2 nodes.

## Configuring and managing the cluster

Clusters must be secured to prevent unauthorized users from connecting to your cluster. It is recommended to use Microsoft Entra ID to authenticate clients, and X.509 certificates for node-to-node security. For more information, see [Service Fabric cluster security scenarios](#).

To configure a public HTTPS endpoint, see [Specify resources in a service manifest](#).

You can scale out the application by adding VMs to the cluster. Virtual machine scale sets support autoscaling using autoscale rules based on performance counters. For more information, see [Scale a Service Fabric cluster in or out using autoscale rules](#).

While the cluster is running, collect logs from all the nodes in a central location. For more information, see [Collect logs by using Azure Diagnostics](#).

## Refactor the application

After the application is ported to Service Fabric, the next step is to refactor it to a more granular architecture. Tailspin's motivation for refactoring is to make it easier to develop, build, and deploy the Surveys application. By decomposing the existing web and worker roles to a more granular architecture, Tailspin wants to remove the existing tightly coupled communication and data dependencies between these roles.

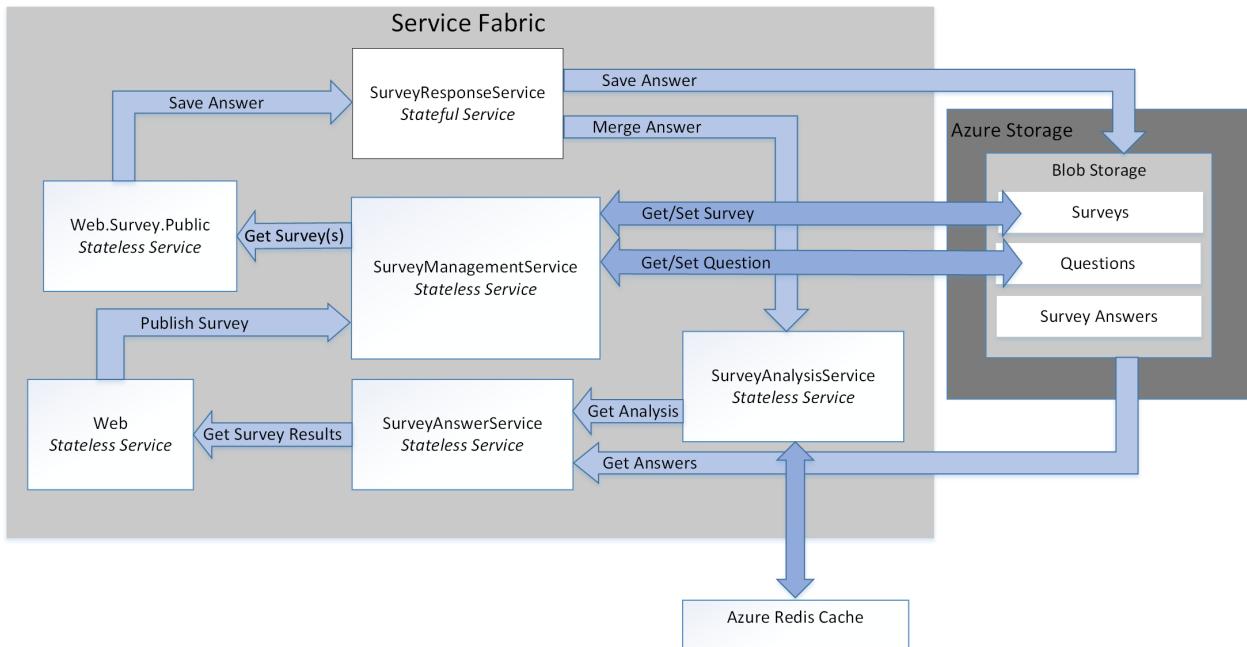
Tailspin sees other benefits in moving the Surveys application to a more granular architecture:

- Each service can be packaged into independent projects with a scope small enough to be managed by a small team.
- Each service can be independently versioned and deployed.
- Each service can be implemented using the best technology for that service. For example, a service fabric cluster can include services built using different versions of the .NET Framework, Java, or other languages such as C or C++.
- Each service can be independently scaled to respond to increases and decreases in load.

The source code for the refactored version of the app is available on [GitHub](#).

## Design considerations

The following diagram shows the architecture of the Surveys application refactored to a more granular architecture:



**Tailspin.Web** is a stateless service self-hosting an ASP.NET MVC application that Tailspin customers visit to create surveys and view survey results. This service shares most of its code with the *Tailspin.Web* service from the ported Service Fabric application. As mentioned earlier, this service uses ASP.NET core and switches from using Kestrel as web frontend to implementing a WebListener.

**Tailspin.Web.Survey.Public** is a stateless service also self-hosting an ASP.NET MVC site. Users visit this site to select surveys from a list and then fill them out. This service shares most of its code with the *Tailspin.Web.Survey.Public* service from the ported Service Fabric application. This service also uses ASP.NET Core and also switches from using Kestrel as web frontend to implementing a WebListener.

**Tailspin.SurveyResponseService** is a stateful service that stores survey answers in Azure Blob Storage. It also merges answers into the survey analysis data. The service is implemented as a stateful service because it uses a [ReliableConcurrentQueue](#) to process survey answers in batches. This functionality was originally implemented in the *Tailspin.AnswerAnalysisService* service in the ported Service Fabric application.

**Tailspin.SurveyManagementService** is a stateless service that stores and retrieves surveys and survey questions. The service uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* and *Tailspin.Web.Survey.Public* services in the ported Service Fabric application. Tailspin refactored the original functionality into this service to allow it to scale independently.

**Tailspin.SurveyAnswerService** is a stateless service that retrieves survey answers and survey analysis. The service also uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* service in the ported Service Fabric application. Tailspin refactored the original functionality into this

service because it expects less load and wants to use fewer instances to conserve resources.

`Tailspin.SurveyAnalysisService` is a stateless service that persists survey answer summary data in a Redis cache for quick retrieval. This service is called by the `Tailspin.SurveyResponseService` each time a survey is answered and the new survey answer data is merged in the summary data. This service includes the functionality originally implemented in the `Tailspin.AnswerAnalysisService` service from the ported Service Fabric application.

## Stateless versus stateful services

Azure Service Fabric supports the following programming models:

- The guest executable model allows any executable to be packaged as a service and deployed to a Service Fabric cluster. Service Fabric orchestrates and manages execution of the guest executable.
- The container model allows for deployment of services in container images. Service Fabric supports creation and management of containers on top of Linux kernel containers as well as Windows Server containers.
- The reliable services programming model allows for the creation of stateless or stateful services that integrate with all Service Fabric platform features. Stateful services allow for replicated state to be stored in the Service Fabric cluster. Stateless services do not.
- The reliable actors programming model allows for the creation of services that implement the virtual actor pattern.

All the services in the Surveys application are stateless reliable services, except for the `Tailspin.SurveyResponseService` service. This service implements a [ReliableConcurrentQueue](#) to process survey answers when they are received. Responses in the ReliableConcurrentQueue are saved into Azure Blob Storage and passed to the `Tailspin.SurveyAnalysisService` for analysis. Tailspin chooses a ReliableConcurrentQueue because responses do not require strict first-in-first-out (FIFO) ordering provided by a queue such as Azure Service Bus. A ReliableConcurrentQueue is also designed to deliver high throughput and low latency for queue and dequeue operations.

Operations to persist dequeued items from a ReliableConcurrentQueue should ideally be idempotent. If an exception is thrown during the processing of an item from the queue, the same item may be processed more than once. In the Surveys application, the operation to merge survey answers to the `Tailspin.SurveyAnalysisService` is not idempotent because Tailspin decided that the survey analysis data is only a current snapshot of the analysis data and does not need to be consistent. The survey answers

saved to Azure Blob Storage are eventually consistent, so the survey final analysis can always be recalculated correctly from this data.

## Communication framework

Each service in the Surveys application communicates using a RESTful web API. RESTful APIs offer the following benefits:

- Ease of use: each service is built using ASP.NET Core MVC, which natively supports the creation of Web APIs.
- Security: While each service does not require SSL, Tailspin could require each service to do so.
- Versioning: clients can be written and tested against a specific version of a web API.

Services in the Survey application use the [reverse proxy](#) implemented by Service Fabric. Reverse proxy is a service that runs on each node in the Service Fabric cluster and provides endpoint resolution, automatic retry, and handles other types of connection failures. To use the reverse proxy, each RESTful API call to a specific service is made using a predefined reverse proxy port. For example, if the reverse proxy port has been set to 19081, a call to the *Tailspin.SurveyAnswerService* can be made as follows:

C#

```
static SurveyAnswerService()
{
 httpClient = new HttpClient
 {
 BaseAddress = new
 Uri("http://localhost:19081/Tailspin/SurveyAnswerService/")
 };
}
```

To enable reverse proxy, specify a reverse proxy port during creation of the Service Fabric cluster. For more information, see [reverse proxy](#) in Azure Service Fabric.

## Performance considerations

Tailspin created the ASP.NET Core services for *Tailspin.Web* and *Tailspin.Web.Surveys.Public* using Visual Studio templates. By default, these templates include logging to the console. Logging to the console may be done during development and debugging, but all logging to the console should be removed when the application is deployed to production.

## ⓘ Note

For more information about setting up monitoring and diagnostics for Service Fabric applications running in production, see [monitoring and diagnostics for Azure Service Fabric](#).

For example, the following lines in *startup.cs* for each of the web front end services should be commented out:

C#

```
// This method gets called by the runtime. Use this method to configure the
// HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
 //loggerFactory.AddConsole(Configuration.GetSection("Logging"));
 //loggerFactory.AddDebug();

 app.UseMvc();
}
```

## ⓘ Note

These lines may be conditionally excluded when Visual Studio is set to **release** when publishing.

Finally, when Tailspin deploys the Tailspin application to production, they switch Visual Studio to **release** mode.

## Deployment considerations

The refactored Surveys application is composed of five stateless services and one stateful service, so cluster planning is limited to determining the correct VM size and number of nodes. In the *applicationmanifest.xml* file that describes the cluster, Tailspin sets the *InstanceCount* attribute of the *StatelessService* tag to -1 for each of the services. A value of -1 directs Service Fabric to create an instance of the service on each node in the cluster.

## ⓘ Note

Stateful services require the additional step of planning the correct number of partitions and replicas for their data.

Tailspin deploys the cluster using the Azure portal. The Service Fabric Cluster resource type deploys all of the necessary infrastructure, including virtual machine scale sets and a load balancer. The recommended VM sizes are displayed in the Azure portal during the provisioning process for the Service Fabric cluster. Because the VMs are deployed in a virtual machine scale set, they can be both scaled up and out as user load increases.

## Next steps

The Surveys application code is available on [GitHub](#).

If you are just getting started with [Azure Service Fabric](#), first set up your development environment then download the latest [Azure SDK](#) and the [Azure Service Fabric SDK](#). The SDK includes the OneBox cluster manager so you can deploy and test the Surveys application locally with full F5 debugging.

## Related resources

- [Migration architecture design](#)
- [Build migration plan with Azure Migrate](#)
- [Modernize enterprise applications with Azure Service Fabric](#)

# Design a microservice-oriented application

Article • 04/13/2022

## 💡 Tip

This content is an excerpt from the eBook, *.NET Microservices Architecture for Containerized .NET Applications*, available on [.NET Docs](#) or as a free downloadable PDF that can be read offline.

[Download PDF](#)



This section focuses on developing a hypothetical server-side enterprise application.

## Application specifications

The hypothetical application handles requests by executing business logic, accessing databases, and then returning HTML, JSON, or XML responses. We will say that the application must support various clients, including desktop browsers running Single Page Applications (SPAs), traditional web apps, mobile web apps, and native mobile apps. The application might also expose an API for third parties to consume. It should also be able to integrate its microservices or external applications asynchronously, so that approach will help resiliency of the microservices in the case of partial failures.

The application will consist of these types of components:

- Presentation components. These components are responsible for handling the UI and consuming remote services.
- Domain or business logic. This component is the application's domain logic.

- Database access logic. This component consists of data access components responsible for accessing databases (SQL or NoSQL).
- Application integration logic. This component includes a messaging channel, based on message brokers.

The application will require high scalability, while allowing its vertical subsystems to scale out autonomously, because certain subsystems will require more scalability than others.

The application must be able to be deployed in multiple infrastructure environments (multiple public clouds and on-premises) and ideally should be cross-platform, able to move from Linux to Windows (or vice versa) easily.

## Development team context

We also assume the following about the development process for the application:

- You have multiple dev teams focusing on different business areas of the application.
- New team members must become productive quickly, and the application must be easy to understand and modify.
- The application will have a long-term evolution and ever-changing business rules.
- You need good long-term maintainability, which means having agility when implementing new changes in the future while being able to update multiple subsystems with minimum impact on the other subsystems.
- You want to practice continuous integration and continuous deployment of the application.
- You want to take advantage of emerging technologies (frameworks, programming languages, etc.) while evolving the application. You do not want to make full migrations of the application when moving to new technologies, because that would result in high costs and impact the predictability and stability of the application.

## Choosing an architecture

What should the application deployment architecture be? The specifications for the application, along with the development context, strongly suggest that you should

architect the application by decomposing it into autonomous subsystems in the form of collaborating [microservices](#) and containers, where a microservice is a container.

In this approach, each service (container) implements a set of cohesive and narrowly related functions. For example, an application might consist of services such as the catalog service, ordering service, basket service, user profile service, etc.

Microservices communicate using protocols such as HTTP (REST), but also asynchronously (for example, using AMQP) whenever possible, especially when propagating updates with integration events.

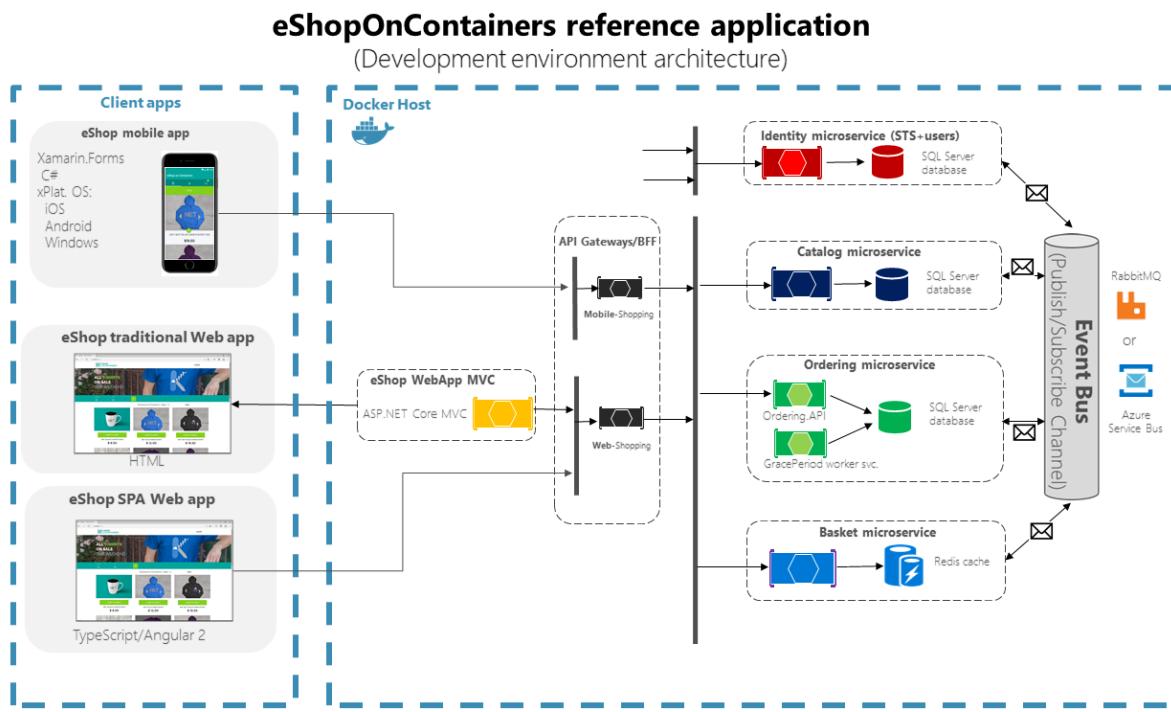
Microservices are developed and deployed as containers independently of one another. This approach means that a development team can be developing and deploying a certain microservice without impacting other subsystems.

Each microservice has its own database, allowing it to be fully decoupled from other microservices. When necessary, consistency between databases from different microservices is achieved using application-level integration events (through a logical event bus), as handled in [Command and Query Responsibility Segregation \(CQRS\)](#). Because of that, the business constraints must embrace eventual consistency between the multiple microservices and related databases.

## eShopOnContainers: A reference application for .NET and microservices deployed using containers

So that you can focus on the architecture and technologies instead of thinking about a hypothetical business domain that you might not know, we have selected a well-known business domain—namely, a simplified e-commerce (e-shop) application that presents a catalog of products, takes orders from customers, verifies inventory, and performs other business functions. This container-based application source code is available in the [eShopOnContainers](#) GitHub repo.

The application consists of multiple subsystems, including several store UI front ends (a Web application and a native mobile app), along with the back-end microservices and containers for all the required server-side operations with several API Gateways as consolidated entry points to the internal microservices. Figure 6-1 shows the architecture of the reference application.



**Figure 6-1.** The eShopOnContainers reference application architecture for development environment

The above diagram shows that Mobile and SPA clients communicate to single API gateway endpoints, that then communicate to microservices. Traditional web clients communicate to MVC microservice, that communicates to microservices through the API gateway.

**Hosting environment.** In Figure 6-1, you see several containers deployed within a single Docker host. That would be the case when deploying to a single Docker host with the docker-compose up command. However, if you are using an orchestrator or container cluster, each container could be running in a different host (node), and any node could be running any number of containers, as we explained earlier in the architecture section.

**Communication architecture.** The eShopOnContainers application uses two communication types, depending on the kind of the functional action (queries versus updates and transactions):

- Http client-to-microservice communication through API Gateways. This approach is used for queries and when accepting update or transactional commands from the client apps. The approach using API Gateways is explained in detail in later sections.
- Asynchronous event-based communication. This communication occurs through an event bus to propagate updates across microservices or to integrate with external applications. The event bus can be implemented with any messaging-broker infrastructure technology like [RabbitMQ](#), or using higher-level

(abstraction-level) service buses like [Azure Service Bus](#), [NServiceBus](#), [MassTransit](#) , or [Brighter](#) .

The application is deployed as a set of microservices in the form of containers. Client apps can communicate with those microservices running as containers through the public URLs published by the API Gateways.

## Data sovereignty per microservice

In the sample application, each microservice owns its own database or data source, although all SQL Server databases are deployed as a single container. This design decision was made only to make it easy for a developer to get the code from GitHub, clone it, and open it in Visual Studio or Visual Studio Code. Or alternatively, it makes it easy to compile the custom Docker images using the .NET CLI and the Docker CLI, and then deploy and run them in a Docker development environment. Either way, using containers for data sources lets developers build and deploy in a matter of minutes without having to provision an external database or any other data source with hard dependencies on infrastructure (cloud or on-premises).

In a real production environment, for high availability and for scalability, the databases should be based on database servers in the cloud or on-premises, but not in containers.

Therefore, the units of deployment for microservices (and even for databases in this application) are Docker containers, and the reference application is a multi-container application that embraces microservices principles.

## Additional resources

- [eShopOnContainers GitHub repo](#). Source code for the reference application  
<https://aka.ms/eShopOnContainers/>

## Benefits of a microservice-based solution

A microservice-based solution like this has many benefits:

**Each microservice is relatively small—easy to manage and evolve.** Specifically:

- It is easy for a developer to understand and get started quickly with good productivity.
- Containers start fast, which makes developers more productive.

- An IDE like Visual Studio can load smaller projects fast, making developers productive.
- Each microservice can be designed, developed, and deployed independently of other microservices, which provide agility because it is easier to deploy new versions of microservices frequently.

**It is possible to scale out individual areas of the application.** For instance, the catalog service or the basket service might need to be scaled out, but not the ordering process. A microservices infrastructure will be much more efficient with regard to the resources used when scaling out than a monolithic architecture would be.

**You can divide the development work between multiple teams.** Each service can be owned by a single development team. Each team can manage, develop, deploy, and scale their service independently of the rest of the teams.

**Issues are more isolated.** If there is an issue in one service, only that service is initially impacted (except when the wrong design is used, with direct dependencies between microservices), and other services can continue to handle requests. In contrast, one malfunctioning component in a monolithic deployment architecture can bring down the entire system, especially when it involves resources, such as a memory leak. Additionally, when an issue in a microservice is resolved, you can deploy just the affected microservice without impacting the rest of the application.

**You can use the latest technologies.** Because you can start developing services independently and run them side by side (thanks to containers and .NET), you can start using the latest technologies and frameworks expediently instead of being stuck on an older stack or framework for the whole application.

## Downsides of a microservice-based solution

A microservice-based solution like this also has some drawbacks:

**Distributed application.** Distributing the application adds complexity for developers when they are designing and building the services. For example, developers must implement inter-service communication using protocols like HTTP or AMQP, which adds complexity for testing and exception handling. It also adds latency to the system.

**Deployment complexity.** An application that has dozens of microservices types and needs high scalability (it needs to be able to create many instances per service and balance those services across many hosts) means a high degree of deployment complexity for IT operations and management. If you are not using a microservice-

oriented infrastructure (like an orchestrator and scheduler), that additional complexity can require far more development efforts than the business application itself.

**Atomic transactions.** Atomic transactions between multiple microservices usually are not possible. The business requirements have to embrace eventual consistency between multiple microservices. For more information, see the [challenges of idempotent message processing](#).

**Increased global resource needs** (total memory, drives, and network resources for all the servers or hosts). In many cases, when you replace a monolithic application with a microservices approach, the amount of initial global resources needed by the new microservice-based application will be larger than the infrastructure needs of the original monolithic application. This approach is because the higher degree of granularity and distributed services requires more global resources. However, given the low cost of resources in general and the benefit of being able to scale out certain areas of the application compared to long-term costs when evolving monolithic applications, the increased use of resources is usually a good tradeoff for large, long-term applications.

**Issues with direct client-to-microservice communication.** When the application is large, with dozens of microservices, there are challenges and limitations if the application requires direct client-to-microservice communications. One problem is a potential mismatch between the needs of the client and the APIs exposed by each of the microservices. In certain cases, the client application might need to make many separate requests to compose the UI, which can be inefficient over the Internet and would be impractical over a mobile network. Therefore, requests from the client application to the back-end system should be minimized.

Another problem with direct client-to-microservice communications is that some microservices might be using protocols that are not Web-friendly. One service might use a binary protocol, while another service might use AMQP messaging. Those protocols are not firewall-friendly and are best used internally. Usually, an application should use protocols such as HTTP and WebSockets for communication outside of the firewall.

Yet another drawback with this direct client-to-service approach is that it makes it difficult to refactor the contracts for those microservices. Over time developers might want to change how the system is partitioned into services. For example, they might merge two services or split a service into two or more services. However, if clients communicate directly with the services, performing this kind of refactoring can break compatibility with client apps.

As mentioned in the architecture section, when designing and building a complex application based on microservices, you might consider the use of multiple fine-grained

API Gateways instead of the simpler direct client-to-microservice communication approach.

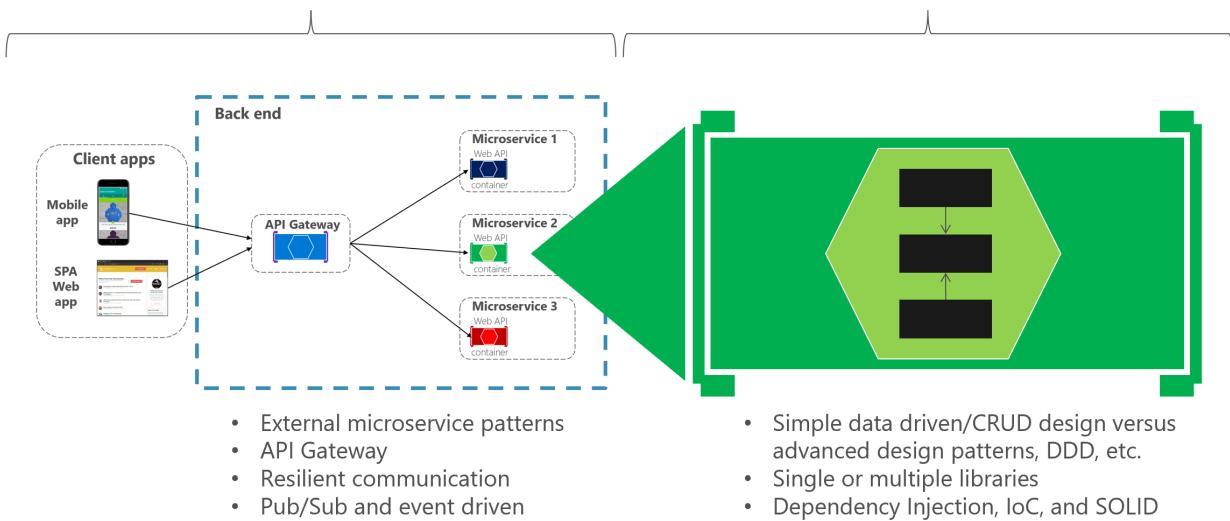
**Partitioning the microservices.** Finally, no matter, which approach you take for your microservice architecture, another challenge is deciding how to partition an end-to-end application into multiple microservices. As noted in the architecture section of the guide, there are several techniques and approaches you can take. Basically, you need to identify areas of the application that are decoupled from the other areas and that have a low number of hard dependencies. In many cases, this approach is aligned to partitioning services by use case. For example, in our e-shop application, we have an ordering service that is responsible for all the business logic related to the order process. We also have the catalog service and the basket service that implement other capabilities. Ideally, each service should have only a small set of responsibilities. This approach is similar to the single responsibility principle (SRP) applied to classes, which states that a class should only have one reason to change. But in this case, it is about microservices, so the scope will be larger than a single class. Most of all, a microservice has to be autonomous, end to end, including responsibility for its own data sources.

## External versus internal architecture and design patterns

The external architecture is the microservice architecture composed by multiple services, following the principles described in the architecture section of this guide. However, depending on the nature of each microservice, and independently of high-level microservice architecture you choose, it is common and sometimes advisable to have different internal architectures, each based on different patterns, for different microservices. The microservices can even use different technologies and programming languages. Figure 6-2 illustrates this diversity.

## External architecture per application

## Internal architecture per microservice



**Figure 6-2.** External versus internal architecture and design

For instance, in our *eShopOnContainers* sample, the catalog, basket, and user profile microservices are simple (basically, CRUD subsystems). Therefore, their internal architecture and design is straightforward. However, you might have other microservices, such as the ordering microservice, which is more complex and represents ever-changing business rules with a high degree of domain complexity. In cases like these, you might want to implement more advanced patterns within a particular microservice, like the ones defined with domain-driven design (DDD) approaches, as we are doing in the *eShopOnContainers* ordering microservice. (We will review these DDD patterns in the section later that explains the implementation of the *eShopOnContainers* ordering microservice.)

Another reason for a different technology per microservice might be the nature of each microservice. For example, it might be better to use a functional programming language like F#, or even a language like R if you are targeting AI and machine learning domains, instead of a more object-oriented programming language like C#.

The bottom line is that each microservice can have a different internal architecture based on different design patterns. Not all microservices should be implemented using advanced DDD patterns, because that would be over-engineering them. Similarly, complex microservices with ever-changing business logic should not be implemented as CRUD components, or you can end up with low-quality code.

## The new world: multiple architectural patterns and polyglot microservices

There are many architectural patterns used by software architects and developers. The following are a few (mixing architecture styles and architecture patterns):

- Simple CRUD, single-tier, single-layer.
- Traditional N-Layered.
- Domain-Driven Design N-layered ↗.
- Clean Architecture (as used with eShopOnWeb ↗)
- Command and Query Responsibility Segregation ↗ (CQRS).
- Event-Driven Architecture ↗ (EDA).

You can also build microservices with many technologies and languages, such as ASP.NET Core Web APIs, NancyFx, ASP.NET Core SignalR (available with .NET Core 2 or later), F#, Node.js, Python, Java, C++, GoLang, and more.

The important point is that no particular architecture pattern or style, nor any particular technology, is right for all situations. Figure 6-3 shows some approaches and technologies (although not in any particular order) that could be used in different microservices.

### The Multi-Architectural-Patterns and polyglot microservices world



**Figure 6-3.** Multi-architectural patterns and the polyglot microservices world

Multi-architectural pattern and polyglot microservices means you can mix and match languages and technologies to the needs of each microservice and still have them

talking to each other. As shown in Figure 6-3, in applications composed of many microservices (Bounded Contexts in domain-driven design terminology, or simply "subsystems" as autonomous microservices), you might implement each microservice in a different way. Each might have a different architecture pattern and use different languages and databases depending on the application's nature, business requirements, and priorities. In some cases, the microservices might be similar. But that is not usually the case, because each subsystem's context boundary and requirements are usually different.

For instance, for a simple CRUD maintenance application, it might not make sense to design and implement DDD patterns. But for your core domain or core business, you might need to apply more advanced patterns to tackle business complexity with ever-changing business rules.

Especially when you deal with large applications composed by multiple subsystems, you should not apply a single top-level architecture based on a single architecture pattern. For instance, CQRS should not be applied as a top-level architecture for a whole application, but might be useful for a specific set of services.

There is no silver bullet or a right architecture pattern for every given case. You cannot have "one architecture pattern to rule them all." Depending on the priorities of each microservice, you must choose a different approach for each, as explained in the following sections.

[Previous](#)[Next](#)

## ⌚ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Creating a simple data-driven CRUD microservice

Article • 03/01/2023

## 💡 Tip

This content is an excerpt from the eBook, *.NET Microservices Architecture for Containerized .NET Applications*, available on [.NET Docs](#) or as a free downloadable PDF that can be read offline.

[Download PDF](#)



This section outlines how to create a simple microservice that performs create, read, update, and delete (CRUD) operations on a data source.

## Designing a simple CRUD microservice

From a design point of view, this type of containerized microservice is very simple. Perhaps the problem to solve is simple, or perhaps the implementation is only a proof of concept.

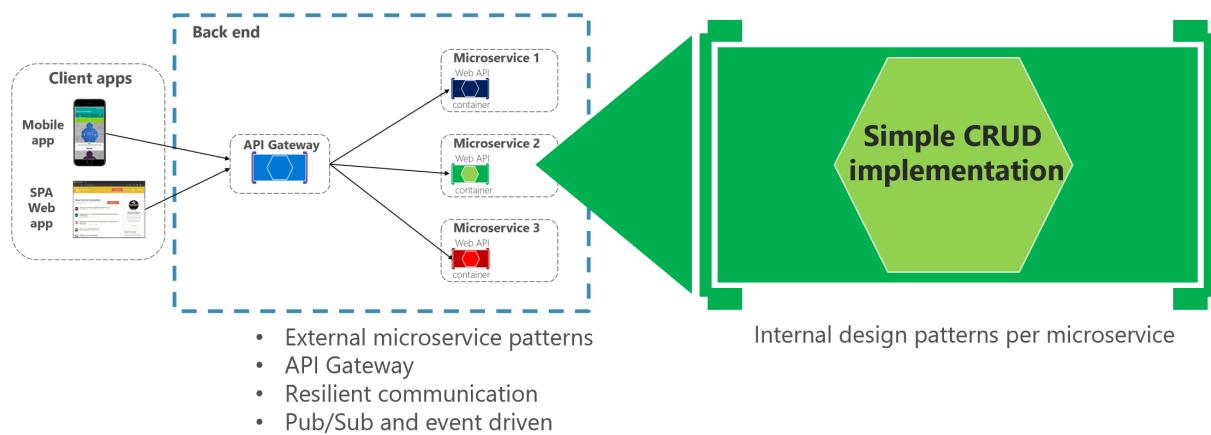


Figure 6-4. Internal design for simple CRUD microservices

An example of this kind of simple data-drive service is the catalog microservice from the eShopOnContainers sample application. This type of service implements all its functionality in a single ASP.NET Core Web API project that includes classes for its data model, its business logic, and its data access code. It also stores its related data in a database running in SQL Server (as another container for dev/test purposes), but could also be any regular SQL Server host, as shown in Figure 6-5.

## Data-Driven/CRUD microservice container

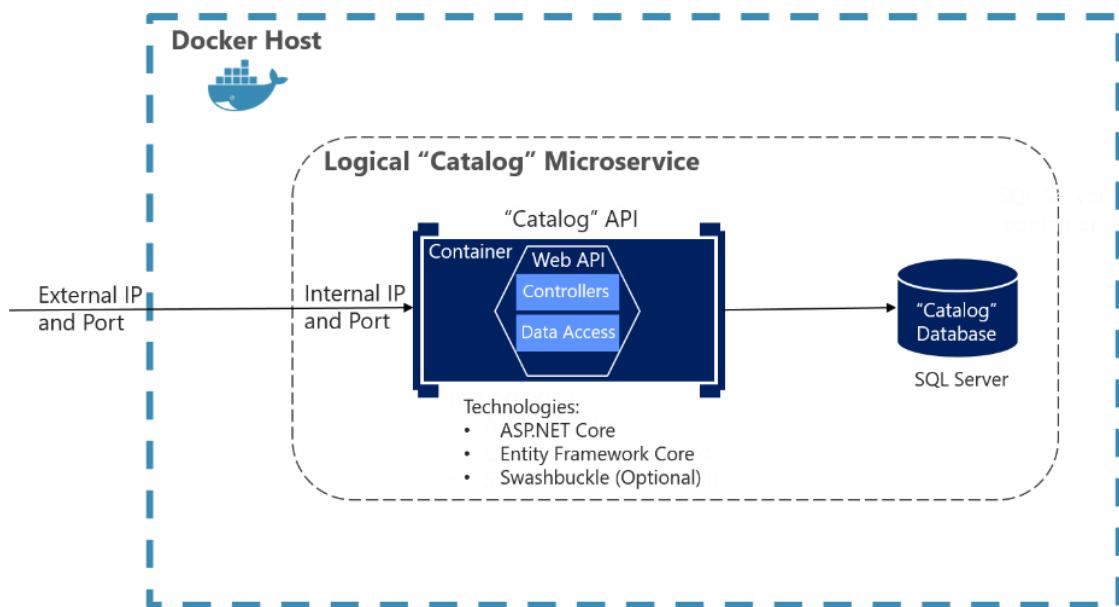


Figure 6-5. Simple data-driven/CRUD microservice design

The previous diagram shows the logical Catalog microservice, that includes its Catalog database, which can be or not in the same Docker host. Having the database in the same Docker host might be good for development, but not for production. When you are developing this kind of service, you only need [ASP.NET Core](#) and a data-access API or ORM like [Entity Framework Core](#). You could also generate [Swagger](#) metadata automatically through [Swashbuckle](#) to provide a description of what your service offers, as explained in the next section.

Note that running a database server like SQL Server within a Docker container is great for development environments, because you can have all your dependencies up and running without needing to provision a database in the cloud or on-premises. This approach is convenient when running integration tests. However, for production environments, running a database server in a container is not recommended, because you usually do not get high availability with that approach. For a production environment in Azure, it is recommended that you use Azure SQL DB or any other

database technology that can provide high availability and high scalability. For example, for a NoSQL approach, you might choose CosmosDB.

Finally, by editing the Dockerfile and docker-compose.yml metadata files, you can configure how the image of this container will be created—what base image it will use, plus design settings such as internal and external names and TCP ports.

## Implementing a simple CRUD microservice with ASP.NET Core

To implement a simple CRUD microservice using .NET and Visual Studio, you start by creating a simple ASP.NET Core Web API project (running on .NET so it can run on a Linux Docker host), as shown in Figure 6-6.

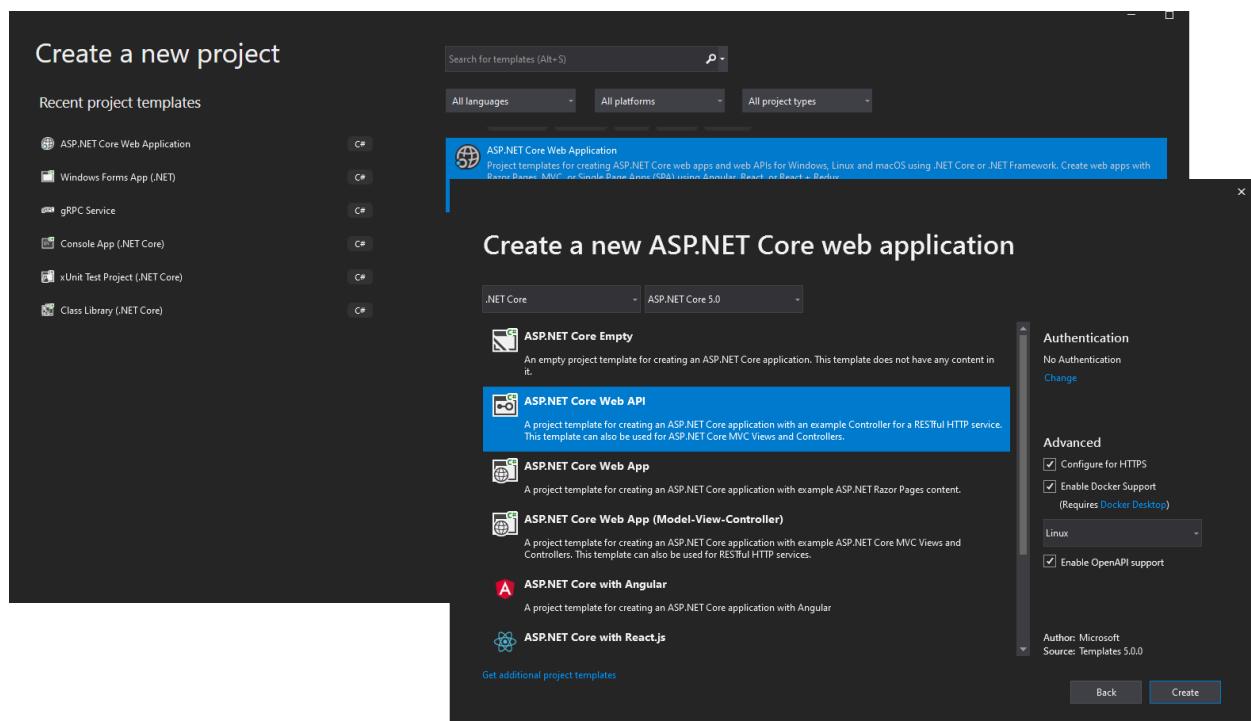


Figure 6-6. Creating an ASP.NET Core Web API project in Visual Studio 2019

To create an ASP.NET Core Web API Project, first select an ASP.NET Core Web Application and then select the API type. After creating the project, you can implement your MVC controllers as you would in any other Web API project, using the Entity Framework API or other API. In a new Web API project, you can see that the only dependency you have in that microservice is on ASP.NET Core itself. Internally, within the *Microsoft.AspNetCore.All* dependency, it is referencing Entity Framework and many other .NET NuGet packages, as shown in Figure 6-7.

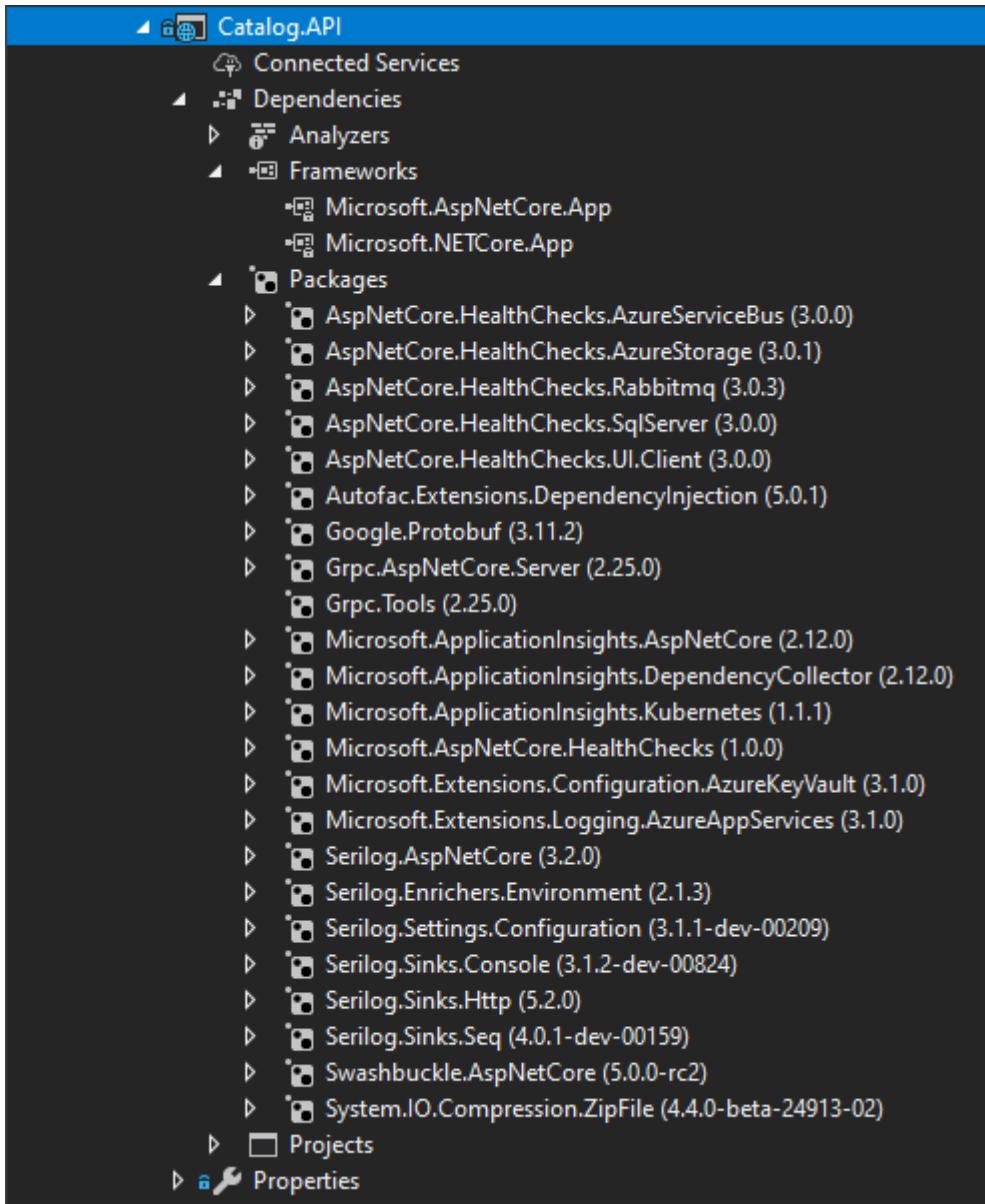


Figure 6-7. Dependencies in a simple CRUD Web API microservice

The API project includes references to Microsoft.AspNetCore.App NuGet package, that includes references to all essential packages. It could include some other packages as well.

## Implementing CRUD Web API services with Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology. EF Core is an object-relational mapper (ORM) that enables .NET developers to work with a database using .NET objects.

The catalog microservice uses EF and the SQL Server provider because its database is running in a container with the SQL Server for Linux Docker image. However, the database could be deployed into any SQL Server, such as Windows on-premises or

Azure SQL DB. The only thing you would need to change is the connection string in the ASP.NET Web API microservice.

## The data model

With EF Core, data access is performed by using a model. A model is made up of (domain model) entity classes and a derived context (DbContext) that represents a session with the database, allowing you to query and save data. You can generate a model from an existing database, manually code a model to match your database, or use EF migrations technique to create a database from your model, using the code-first approach (that makes it easy to evolve the database as your model changes over time). For the catalog microservice, the last approach has been used. You can see an example of the CatalogItem entity class in the following code example, which is a simple Plain Old Class Object (POCO) entity class.

C#

```
public class CatalogItem
{
 public int Id { get; set; }
 public string Name { get; set; }
 public string Description { get; set; }
 public decimal Price { get; set; }
 public string PictureFileName { get; set; }
 public string PictureUri { get; set; }
 public int CatalogTypeId { get; set; }
 public CatalogType CatalogType { get; set; }
 public int CatalogBrandId { get; set; }
 public CatalogBrand CatalogBrand { get; set; }
 public int AvailableStock { get; set; }
 public int RestockThreshold { get; set; }
 public int MaxStockThreshold { get; set; }

 public bool OnReorder { get; set; }
 public CatalogItem() { }

 // Additional code ...
}
```

You also need a DbContext that represents a session with the database. For the catalog microservice, the CatalogContext class derives from the DbContext base class, as shown in the following example:

C#

```
public class CatalogContext : DbContext
{
```

```

 public CatalogContext(DbContextOptions<CatalogContext> options) :
base(options)
 { }
 public DbSet<CatalogItem> CatalogItems { get; set; }
 public DbSet<CatalogBrand> CatalogBrands { get; set; }
 public DbSet<CatalogType> CatalogTypes { get; set; }

 // Additional code ...
}

```

You can have additional `DbContext` implementations. For example, in the sample Catalog.API microservice, there's a second `DbContext` named `CatalogContextSeed` where it automatically populates the sample data the first time it tries to access the database. This method is useful for demo data and for automated testing scenarios, as well.

Within the `DbContext`, you use the `OnModelCreating` method to customize object/database entity mappings and other [EF extensibility points](#).

## Querying data from Web API controllers

Instances of your entity classes are typically retrieved from the database using Language-Integrated Query (LINQ), as shown in the following example:

C#

```

[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
 private readonly CatalogContext _catalogContext;
 private readonly CatalogSettings _settings;
 private readonly ICatalogIntegrationEventService
 _catalogIntegrationEventService;

 public CatalogController(
 CatalogContext context,
 IOptionsSnapshot<CatalogSettings> settings,
 ICatalogIntegrationEventService catalogIntegrationEventService)
 {
 _catalogContext = context ?? throw new
ArgumentNullException(nameof(context));
 _catalogIntegrationEventService = catalogIntegrationEventService
 ?? throw new
ArgumentNullException(nameof(catalogIntegrationEventService));

 _settings = settings.Value;
 context.ChangeTracker.QueryTrackingBehavior =
QueryTrackingBehavior.NoTracking;
 }

 // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]

```

```

[HttpGet]
[Route("items")]
[ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>),
(int) HttpStatusCode.OK)]
[ProducesResponseType(typeof(IEnumerable<CatalogItem>),
(int) HttpStatusCode.OK)]
[ProducesResponseType((int) HttpStatusCode.BadRequest)]
public async Task<IActionResult> ItemsAsync(
 [FromQuery] int pageSize = 10,
 [FromQuery] int pageIndex = 0,
 string ids = null)
{
 if (!string.IsNullOrEmpty(ids))
 {
 var items = await GetItemsByIdsAsync(ids);

 if (!items.Any())
 {
 return BadRequest("ids value invalid. Must be comma-separated list of numbers");
 }

 return Ok(items);
 }

 var totalItems = await _catalogContext.CatalogItems
 .LongCountAsync();

 var itemsOnPage = await _catalogContext.CatalogItems
 .OrderBy(c => c.Name)
 .Skip(pageSize * pageIndex)
 .Take(pageSize)
 .ToListAsync();

 itemsOnPage = ChangeUriPlaceholder(itemsOnPage);

 var model = new PaginatedItemsViewModel<CatalogItem>(
 pageIndex, pageSize, totalItems, itemsOnPage);

 return Ok(model);
}
//...
}

```

## Saving data

Data is created, deleted, and modified in the database using instances of your entity classes. You could add code like the following hard-coded example (mock data, in this case) to your Web API controllers.

```
var catalogItem = new CatalogItem() {CatalogTypeId=2, CatalogBrandId=2,
 Name="Roslyn T-Shirt", Price = 12};
_context.Catalog.Add(catalogItem);
_context.SaveChanges();
```

## Dependency Injection in ASP.NET Core and Web API controllers

In ASP.NET Core, you can use Dependency Injection (DI) out of the box. You do not need to set up a third-party Inversion of Control (IoC) container, although you can plug your preferred IoC container into the ASP.NET Core infrastructure if you want. In this case, it means that you can directly inject the required EF DBContext or additional repositories through the controller constructor.

In the `CatalogController` class mentioned earlier, `CatalogContext` (which inherits from `DbContext`) type is injected along with the other required objects in the `CatalogController()` constructor.

An important configuration to set up in the Web API project is the `DbContext` class registration into the service's IoC container. You typically do so in the `Program.cs` file by calling the `builder.Services.AddDbContext<CatalogContext>()` method, as shown in the following **simplified** example:

C#

```
// Additional code...

builder.Services.AddDbContext<CatalogContext>(options =>
{
 options.UseSqlServer(builder.Configuration["ConnectionString"],
 sqlServerOptionsAction: sqlOptions =>
 {
 sqlOptions.MigrationsAssembly(
 typeof(Program).GetTypeInfo().Assembly.GetName().Name);

 //Configuring Connection Resiliency:
 sqlOptions.
 EnableRetryOnFailure(maxRetryCount: 5,
 maxRetryDelay: TimeSpan.FromSeconds(30),
 errorNumbersToAdd: null);
 });

 // Changing default behavior when client evaluation occurs to throw.
 // Default in EFCore would be to log warning when client evaluation is
 // done.
 options.ConfigureWarnings(warnings => warnings.Throw(
```

```
 RelationalEventId.QueryClientEvaluationWarning));
});
```

## Additional resources

- **Querying Data**  
<https://learn.microsoft.com/ef/core/querying/index>
- **Saving Data**  
<https://learn.microsoft.com/ef/core/saving/index>

## The DB connection string and environment variables used by Docker containers

You can use the ASP.NET Core settings and add a `ConnectionString` property to your `settings.json` file as shown in the following example:

JSON

```
{
 "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=
[PLACEHOLDER]",
 "ExternalCatalogBaseUrl": "http://host.docker.internal:5101",
 "Logging": {
 "IncludeScopes": false,
 "LogLevel": {
 "Default": "Debug",
 "System": "Information",
 "Microsoft": "Information"
 }
 }
}
```

The `settings.json` file can have default values for the `ConnectionString` property or for any other property. However, those properties will be overridden by the values of environment variables that you specify in the `docker-compose.override.yml` file, when using Docker.

From your `docker-compose.yml` or `docker-compose.override.yml` files, you can initialize those environment variables so that Docker will set them up as OS environment variables for you, as shown in the following `docker-compose.override.yml` file (the connection string and other lines wrap in this example, but it would not wrap in your own file).

```
yml
```

```
docker-compose.override.yml

#
catalog-api:
 environment:
 -
 ConnectionString=Server=sqldata;Database=Microsoft.eShopOnContainers.Service
 s.CatalogDb;User Id=sa;Password=[PLACEHOLDER]
 # Additional environment variables for this service
 ports:
 - "5101:80"
```

The docker-compose.yml files at the solution level are not only more flexible than configuration files at the project or microservice level, but also more secure if you override the environment variables declared at the docker-compose files with values set from your deployment tools, like from Azure DevOps Services Docker deployment tasks.

Finally, you can get that value from your code by using `builder.Configuration\[ "ConnectionString" \]`, as shown in an earlier code example.

However, for production environments, you might want to explore additional ways on how to store secrets like the connection strings. An excellent way to manage application secrets is using [Azure Key Vault](#).

Azure Key Vault helps to store and safeguard cryptographic keys and secrets used by your cloud applications and services. A secret is anything you want to keep strict control of, like API keys, connection strings, passwords, etc. and strict control includes usage logging, setting expiration, managing access, *among others*.

Azure Key Vault allows a detailed control level of the application secrets usage without the need to let anyone know them. The secrets can even be rotated for enhanced security without disrupting development or operations.

Applications have to be registered in the organization's Active Directory, so they can use the Key Vault.

You can check the [Key Vault Concepts documentation](#) for more details.

## Implementing versioning in ASP.NET Web APIs

As business requirements change, new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. Updating a Web API to handle new requirements is a

relatively straightforward process, but you must consider the effects that such changes will have on client applications consuming the Web API. Although the developer designing and implementing a Web API has full control over that API, the developer does not have the same degree of control over client applications that might be built by third-party organizations operating remotely.

Versioning enables a Web API to indicate the features and resources that it exposes. A client application can then submit requests to a specific version of a feature or resource. There are several approaches to implement versioning:

- URI versioning
- Query string versioning
- Header versioning

Query string and URI versioning are the simplest to implement. Header versioning is a good approach. However, header versioning is not as explicit and straightforward as URI versioning. Because URL versioning is the simplest and most explicit, the eShopOnContainers sample application uses URI versioning.

With URI versioning, as in the eShopOnContainers sample application, each time you modify the Web API or change the schema of resources, you add a version number to the URI for each resource. Existing URIs should continue to operate as before, returning resources that conform to the schema that matches the requested version.

As shown in the following code example, the version can be set by using the `Route` attribute in the Web API controller, which makes the version explicit in the URI (v1 in this case).

```
C#

[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
 // Implementation ...
}
```

This versioning mechanism is simple and depends on the server routing the request to the appropriate endpoint. However, for a more sophisticated versioning and the best method when using REST, you should use hypermedia and implement [HATEOAS](#) (Hypertext as the Engine of Application State).

## Additional resources

- **ASP.NET API Versioning** \ <https://github.com/dotnet/aspnet-api-versioning> ↗
- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy**  
<https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx> ↗
- **Versioning a RESTful web API**  
<https://learn.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api>
- **Roy Fielding. Versioning, Hypermedia, and REST**  
<https://www.infoq.com/articles/roy-fielding-on-versioning> ↗

## Generating Swagger description metadata from your ASP.NET Core Web API

[Swagger](#) ↗ is a commonly used open source framework backed by a large ecosystem of tools that helps you design, build, document, and consume your RESTful APIs. It is becoming the standard for the APIs description metadata domain. You should include Swagger description metadata with any kind of microservice, either data-driven microservices or more advanced domain-driven microservices (as explained in the following section).

The heart of Swagger is the Swagger specification, which is API description metadata in a JSON or YAML file. The specification creates the RESTful contract for your API, detailing all its resources and operations in both a human- and machine-readable format for easy development, discovery, and integration.

The specification is the basis of the OpenAPI Specification (OAS) and is developed in an open, transparent, and collaborative community to standardize the way RESTful interfaces are defined.

The specification defines the structure for how a service can be discovered and how its capabilities understood. For more information, including a web editor and examples of Swagger specifications from companies like Spotify, Uber, Slack, and Microsoft, see the Swagger site (<https://swagger.io> ↗).

## Why use Swagger?

The main reasons to generate Swagger metadata for your APIs are the following.

**Ability for other products to automatically consume and integrate your APIs.** Dozens of products and [commercial tools](#) and many [libraries and frameworks](#) support Swagger. Microsoft has high-level products and tools that can automatically consume Swagger-based APIs, such as the following:

- [AutoRest](#). You can automatically generate .NET client classes for calling Swagger. This tool can be used from the CLI and it also integrates with Visual Studio for easy use through the GUI.
- [Microsoft Flow](#). You can automatically [use and integrate your API](#) into a high-level Microsoft Flow workflow, with no programming skills required.
- [Microsoft PowerApps](#). You can automatically consume your API from [PowerApps mobile apps](#) built with [PowerApps Studio](#), with no programming skills required.
- [Azure App Service Logic Apps](#). You can automatically [use and integrate your API](#) into an [Azure App Service Logic App](#), with no programming skills required.

**Ability to automatically generate API documentation.** When you create large-scale RESTful APIs, such as complex microservice-based applications, you need to handle many endpoints with different data models used in the request and response payloads. Having proper documentation and having a solid API explorer, as you get with Swagger, is key for the success of your API and adoption by developers.

Swagger's metadata is what Microsoft Flow, PowerApps, and Azure Logic Apps use to understand how to use APIs and connect to them.

There are several options to automate Swagger metadata generation for ASP.NET Core REST API applications, in the form of functional API help pages, based on *swagger-ui*.

Probably the best known is [Swashbuckle](#), which is currently used in [eShopOnContainers](#) and we'll cover in some detail in this guide but there's also the option to use [NSwag](#), which can generate Typescript and C# API clients, as well as C# controllers, from a Swagger or OpenAPI specification and even by scanning the .dll that contains the controllers, using [NSwagStudio](#).

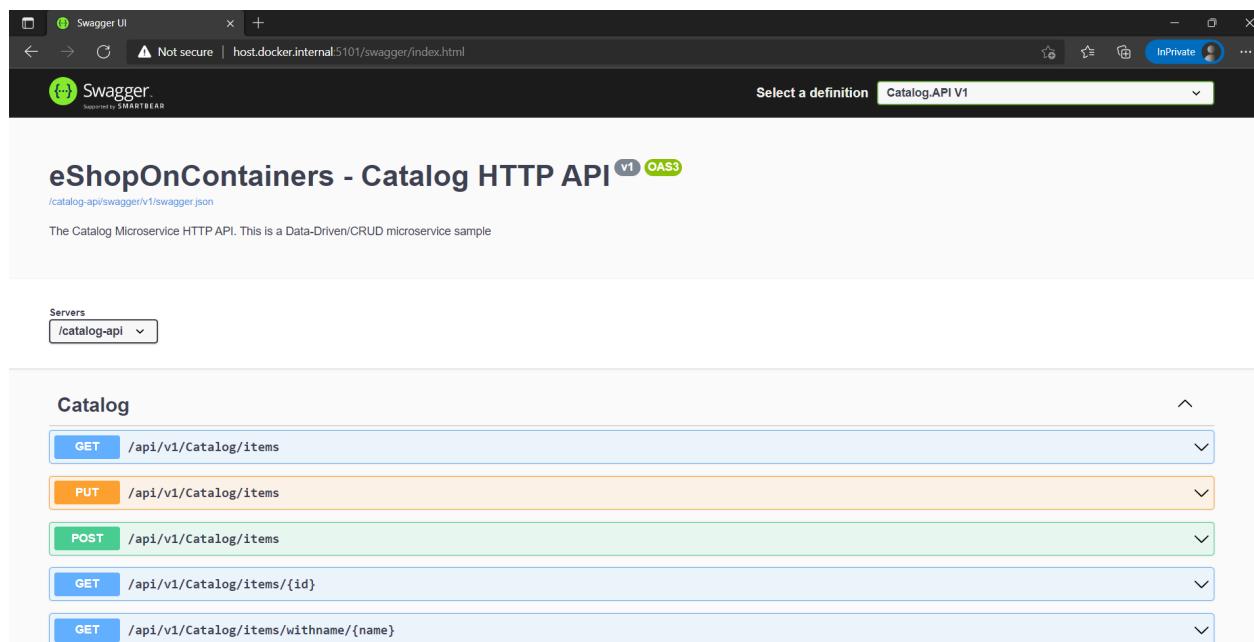
## How to automate API Swagger metadata generation with the Swashbuckle NuGet package

Generating Swagger metadata manually (in a JSON or YAML file) can be tedious work. However, you can automate API discovery of ASP.NET Web API services by using the [Swashbuckle NuGet package](#) to dynamically generate Swagger API metadata.

Swashbuckle automatically generates Swagger metadata for your ASP.NET Web API projects. It supports ASP.NET Core Web API projects and the traditional ASP.NET Web API and any other flavor, such as Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET. It also supports plain Web API deployed on containers, as in for the reference application.

Swashbuckle combines API Explorer and Swagger or [swagger-ui](#) to provide a rich discovery and documentation experience for your API consumers. In addition to its Swagger metadata generator engine, Swashbuckle also contains an embedded version of `swagger-ui`, which it will automatically serve up once Swashbuckle is installed.

This means you can complement your API with a nice discovery UI to help developers to use your API. It requires a small amount of code and maintenance because it is automatically generated, allowing you to focus on building your API. The result for the API Explorer looks like Figure 6-8.



**Figure 6-8.** Swashbuckle API Explorer based on Swagger metadata—eShopOnContainers catalog microservice

The Swashbuckle generated Swagger UI API documentation includes all published actions. The API explorer is not the most important thing here. Once you have a Web API that can describe itself in Swagger metadata, your API can be used seamlessly from Swagger-based tools, including client proxy-class code generators that can target many platforms. For example, as mentioned, [AutoRest](#) automatically generates .NET client classes. But additional tools like [swagger-codegen](#) are also available, which allow code generation of API client libraries, server stubs, and documentation automatically.

Currently, Swashbuckle consists of five internal NuGet packages under the high-level metapackage [Swashbuckle.AspNetCore](#) for ASP.NET Core applications.

After you have installed these NuGet packages in your Web API project, you need to configure Swagger in the *Program.cs* class, as in the following **simplified** code:

```
C#

// Add framework services.

builder.Services.AddSwaggerGen(options =>
{
 options.DescribeAllEnumsAsStrings();
 options.SwaggerDoc("v1", new OpenApiInfo
 {
 Title = "eShopOnContainers - Catalog HTTP API",
 Version = "v1",
 Description = "The Catalog Microservice HTTP API. This is a Data-
 Driven/CRUD microservice sample"
 });
});

// Other startup code...

app.UseSwagger()
 .UseSwaggerUI(c =>
{
 c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
});
...

Once this is done, you can start your application and browse the following
Swagger JSON and UI endpoints using URLs like these:
```

```
```console  
http://<your-root-url>/swagger/v1/swagger.json  
  
http://<your-root-url>/swagger/
```

You previously saw the generated UI created by Swashbuckle for a URL like `http://<your-root-url>/swagger`. In Figure 6-9, you can also see how you can test any API method.

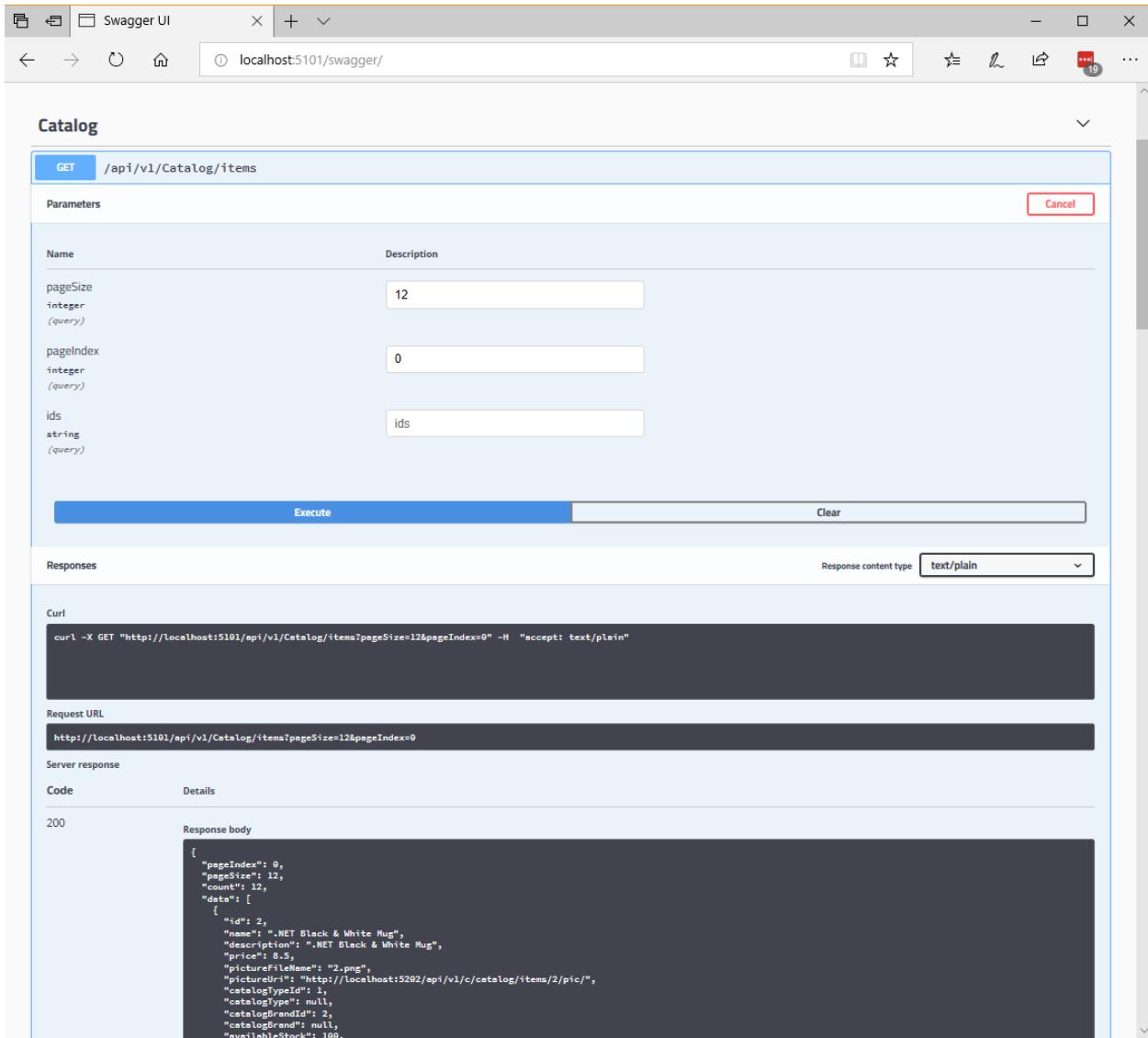


Figure 6-9. Swashbuckle UI testing the Catalog/Items API method

The Swagger UI API detail shows a sample of the response and can be used to execute the real API, which is great for developer discovery. Figure 6-10 shows the Swagger JSON metadata generated from the eShopOnContainers microservice (which is what the tools use underneath) when you request `http://<your-root-url>/swagger/v1/swagger.json` using [Postman](#).

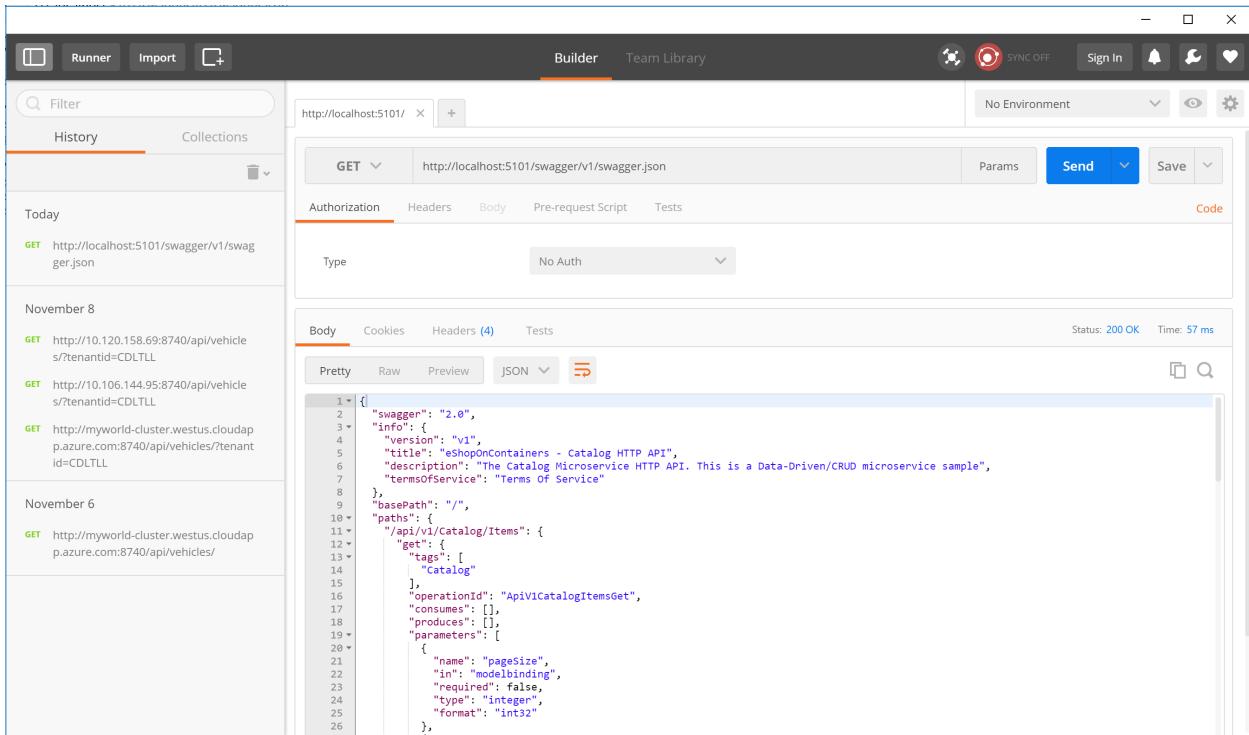


Figure 6-10. Swagger JSON metadata

It is that simple. And because it is automatically generated, the Swagger metadata will grow when you add more functionality to your API.

Additional resources

- **ASP.NET Web API Help Pages using Swagger**
<https://learn.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger>
- **Get started with Swashbuckle and ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle>
- **Get started with NSwag and ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag>

[Previous](#)

[Next](#)

Implementing event-based communication between microservices (integration events)

Article • 03/01/2023

Tip

This content is an excerpt from the eBook, [.NET Microservices Architecture for Containerized .NET Applications](#), available on [.NET Docs](#) or as a free downloadable PDF that can be read offline.

[Download PDF](#)



As described earlier, when you use [event-based communication](#), a [microservice](#) publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event, it can update its own business entities, which might lead to more events being published. This is the essence of the eventual consistency concept. This [publish/subscribe](#) system is usually performed by using an implementation of an event bus. The event bus can be designed as an interface with the API needed to subscribe and unsubscribe to events and to publish events. It can also have one or more implementations based on any inter-process or messaging communication, such as a messaging queue or a service bus that supports asynchronous communication and a publish/subscribe model.

You can use events to implement business transactions that span multiple services, which give you eventual consistency between those services. An eventually consistent transaction consists of a series of distributed actions. At each action, the microservice updates a business entity and publishes an event that triggers the next action. Be aware that transaction do not span the underlying persistence and event bus, so [idempotence](#)

needs to be handled. Figure 6-18 below, shows a PriceUpdated event published through an event bus, so the price update is propagated to the Basket and other microservices.

Implementing asynchronous event-driven communication with an event bus

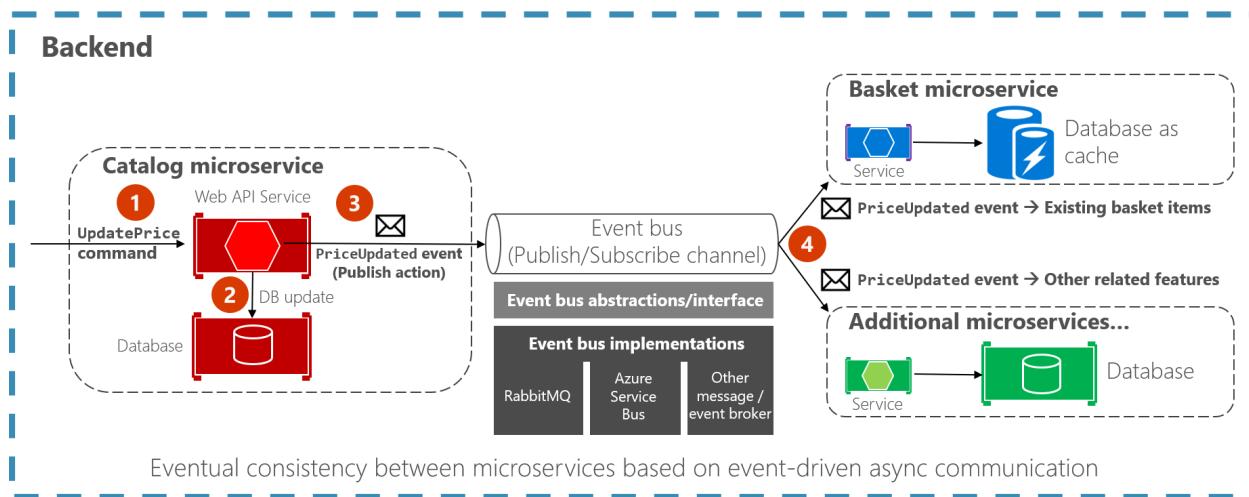


Figure 6-18. Event-driven communication based on an event bus

This section describes how you can implement this type of communication with .NET by using a generic event bus interface, as shown in Figure 6-18. There are multiple potential implementations, each using a different technology or infrastructure such as RabbitMQ, Azure Service Bus, or any other third-party open-source or commercial service bus.

Using message brokers and service buses for production systems

As noted in the architecture section, you can choose from multiple messaging technologies for implementing your abstract event bus. But these technologies are at different levels. For instance, RabbitMQ, a messaging broker transport, is at a lower level than commercial products like Azure Service Bus, NServiceBus, MassTransit, or Brighter. Most of these products can work on top of either RabbitMQ or Azure Service Bus. Your choice of product depends on how many features and how much out-of-the-box scalability you need for your application.

For implementing just an event bus proof-of-concept for your development environment, as in the eShopOnContainers sample, a simple implementation on top of [RabbitMQ](#) running as a container might be enough. But for mission-critical and production systems that need high scalability, you might want to evaluate and use [Azure Service Bus](#).

If you require high-level abstractions and richer features like [Sagas](#) for long-running processes that make distributed development easier, other commercial and open-source service buses like [NServiceBus](#), [MassTransit](#), and [Brighter](#) are worth evaluating. In this case, the abstractions and API to use would usually be directly the ones provided by those high-level service buses instead of your own abstractions (like the [simple event bus abstractions](#) provided at [eShopOnContainers](#)). For that matter, you can research the forked [eShopOnContainers](#) using [NServiceBus](#) (additional derived sample implemented by Particular Software).

Of course, you could always build your own service bus features on top of lower-level technologies like RabbitMQ and Docker, but the work needed to "reinvent the wheel" might be too costly for a custom enterprise application.

To reiterate: the sample event bus abstractions and implementation showcased in the [eShopOnContainers](#) sample are intended to be used only as a proof of concept. Once you have decided that you want to have asynchronous and event-driven communication, as explained in the current section, you should choose the service bus product that best fits your needs for production.

Integration events

Integration events are used for bringing domain state in sync across multiple microservices or external systems. This functionality is done by publishing integration events outside the microservice. When an event is published to multiple receiver microservices (to as many microservices as are subscribed to the integration event), the appropriate event handler in each receiver microservice handles the event.

An integration event is basically a data-holding class, as in the following example:

C#

```
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal
newPrice,
        decimal oldPrice)
    {
        ProductId = productId;
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

```
    }  
}
```

The integration events can be defined at the application level of each microservice, so they are decoupled from other microservices, in a way comparable to how ViewModels are defined in the server and client. What is not recommended is sharing a common integration events library across multiple microservices; doing that would be coupling those microservices with a single event definition data library. You do not want to do that for the same reasons that you do not want to share a common domain model across multiple microservices: microservices must be completely autonomous. For more information, see this blog post on [the amount of data to put in events](#). Be careful not to take this too far, as this other blog post describes [the problem data deficient messages can produce](#). Your design of your events should aim to be "just right" for the needs of their consumers.

There are only a few kinds of libraries you should share across microservices. One is libraries that are final application blocks, like the [Event Bus client API](#), as in eShopOnContainers. Another is libraries that constitute tools that could also be shared as NuGet components, like JSON serializers.

The event bus

An event bus allows publish/subscribe-style communication between microservices without requiring the components to explicitly be aware of each other, as shown in Figure 6-19.

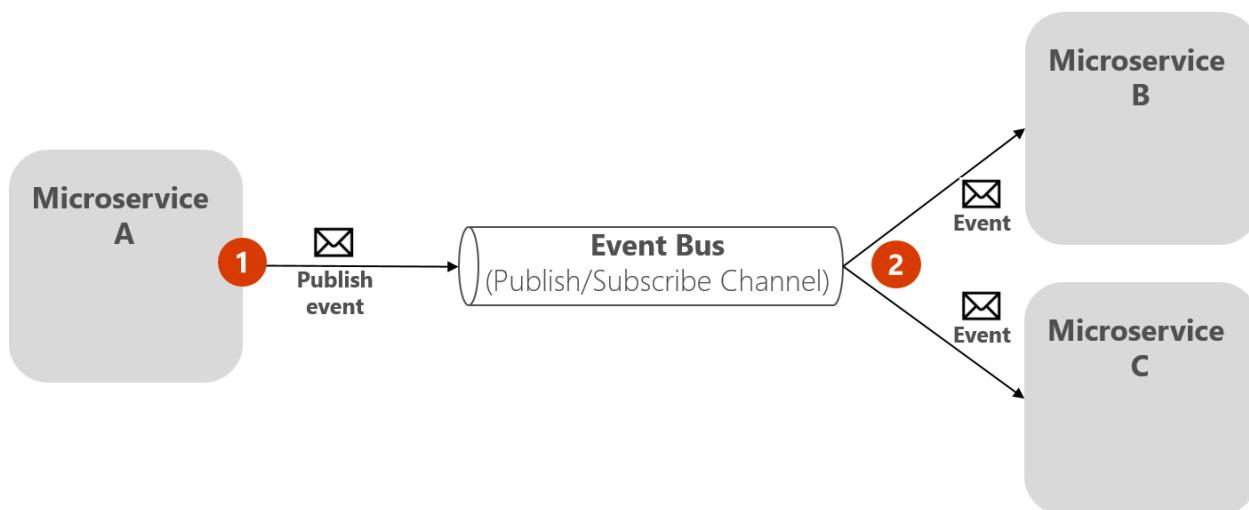


Figure 6-19. Publish/subscribe basics with an event bus

The above diagram shows that microservice A publishes to Event Bus, which distributes to subscribing microservices B and C, without the publisher needing to know the

subscribers. The event bus is related to the Observer pattern and the publish-subscribe pattern.

Observer pattern

In the [Observer pattern](#), your primary object (known as the Observable) notifies other interested objects (known as Observers) with relevant information (events).

Publish/Subscribe (Pub/Sub) pattern

The purpose of the [Publish/Subscribe pattern](#) is the same as the Observer pattern: you want to notify other services when certain events take place. But there is an important difference between the Observer and Pub/Sub patterns. In the observer pattern, the broadcast is performed directly from the observable to the observers, so they "know" each other. But when using a Pub/Sub pattern, there is a third component, called broker, or message broker or event bus, which is known by both the publisher and subscriber. Therefore, when using the Pub/Sub pattern the publisher and the subscribers are precisely decoupled thanks to the mentioned event bus or message broker.

The middleman or event bus

How do you achieve anonymity between publisher and subscriber? An easy way is let a middleman take care of all the communication. An event bus is one such middleman.

An event bus is typically composed of two parts:

- The abstraction or interface.
- One or more implementations.

In Figure 6-19 you can see how, from an application point of view, the event bus is nothing more than a Pub/Sub channel. The way you implement this asynchronous communication can vary. It can have multiple implementations so that you can swap between them, depending on the environment requirements (for example, production versus development environments).

In Figure 6-20, you can see an abstraction of an event bus with multiple implementations based on infrastructure messaging technologies like RabbitMQ, Azure Service Bus, or another event/message broker.



Figure 6- 20. Multiple implementations of an event bus

It's good to have the event bus defined through an interface so it can be implemented with several technologies, like RabbitMQ, Azure Service bus or others. However, and as mentioned previously, using your own abstractions (the event bus interface) is good only if you need basic event bus features supported by your abstractions. If you need richer service bus features, you should probably use the API and abstractions provided by your preferred commercial service bus instead of your own abstractions.

Defining an event bus interface

Let's start with some implementation code for the event bus interface and possible implementations for exploration purposes. The interface should be generic and straightforward, as in the following interface.

C#

```
public interface IEventBus
{
    void Publish(IntegrationEvent @event);

    void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIIntegrationEventHandler<T>;

    void SubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void UnsubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void Unsubscribe<T, TH>()
        where TH : IIIntegrationEventHandler<T>
}
```

```
    where T : IntegrationEvent;  
}
```

The `Publish` method is straightforward. The event bus will broadcast the integration event passed to it to any microservice, or even an external application, subscribed to that event. This method is used by the microservice that is publishing the event.

The `Subscribe` methods (you can have several implementations depending on the arguments) are used by the microservices that want to receive events. This method has two arguments. The first is the integration event to subscribe to (`IntegrationEvent`). The second argument is the integration event handler (or callback method), named `IIntegrationEventHandler<T>`, to be executed when the receiver microservice gets that integration event message.

Additional resources

Some production-ready messaging solutions:

- **Azure Service Bus**
<https://learn.microsoft.com/azure/service-bus-messaging/>
- **NServiceBus**
[https://particular.net/nservicebus ↗](https://particular.net/nservicebus)
- **MassTransit**
[https://masstransit-project.com/ ↗](https://masstransit-project.com/)

[Previous](#)

[Next](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Implement API Gateways with Ocelot

Article • 03/01/2023

Tip

This content is an excerpt from the eBook, .NET Microservices Architecture for Containerized .NET Applications, available on [.NET Docs](#) or as a free downloadable PDF that can be read offline.

[Download PDF](#)



Important

The reference microservice application [eShopOnContainers](#) is currently using features provided by [Envoy](#) to implement the API Gateway instead of the earlier referenced [Ocelot](#). We made this design choice because of Envoy's built-in support for the WebSocket protocol, required by the new gRPC inter-service communications implemented in eShopOnContainers. However, we've retained this section in the guide so you can consider Ocelot as a simple, capable, and lightweight API Gateway suitable for production-grade scenarios. Also, latest Ocelot version contains a breaking change on its json schema. Consider using Ocelot < v16.0.0, or use the key Routes instead of ReRoutes.

Architect and design your API Gateways

The following architecture diagram shows how API Gateways were implemented with Ocelot in eShopOnContainers.

eShopOnContainers reference application (Development environment architecture)

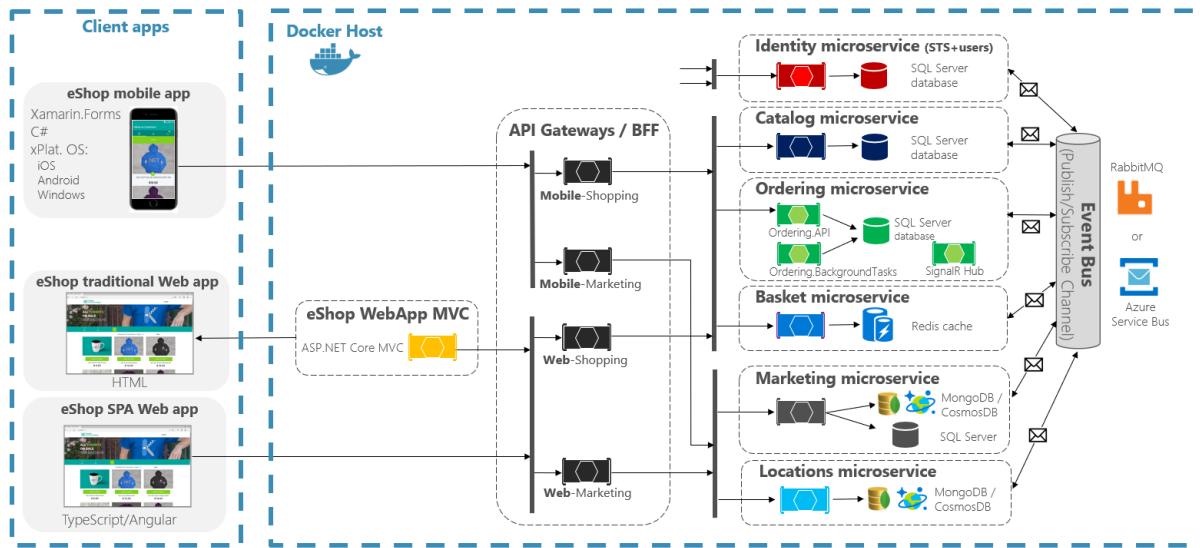


Figure 6-28. eShopOnContainers architecture with API Gateways

That diagram shows how the whole application is deployed into a single Docker host or development PC with "Docker for Windows" or "Docker for Mac". However, deploying into any orchestrator would be similar, but any container in the diagram could be scaled out in the orchestrator.

In addition, the infrastructure assets such as databases, cache, and message brokers should be offloaded from the orchestrator and deployed into high available systems for infrastructure, like Azure SQL Database, Azure Cosmos DB, Azure Redis, Azure Service Bus, or any HA clustering solution on-premises.

As you can also notice in the diagram, having several API Gateways allows multiple development teams to be autonomous (in this case Marketing features vs. Shopping features) when developing and deploying their microservices plus their own related API Gateways.

If you had a single monolithic API Gateway that would mean a single point to be updated by several development teams, which could couple all the microservices with a single part of the application.

Going much further in the design, sometimes a fine-grained API Gateway can also be limited to a single business microservice depending on the chosen architecture. Having the API Gateway's boundaries dictated by the business or domain will help you to get a better design.

For instance, fine granularity in the API Gateway tier can be especially useful for more advanced composite UI applications that are based on microservices, because the concept of a fine-grained API Gateway is similar to a UI composition service.

We delve into more details in the previous section [Creating composite UI based on microservices](#).

As a key takeaway, for many medium- and large-size applications, using a custom-built API Gateway product is usually a good approach, but not as a single monolithic aggregator or unique central custom API Gateway unless that API Gateway allows multiple independent configuration areas for the several development teams creating autonomous microservices.

Sample microservices/containers to reroute through the API Gateways

As an example, eShopOnContainers has around six internal microservice-types that have to be published through the API Gateways, as shown in the following image.

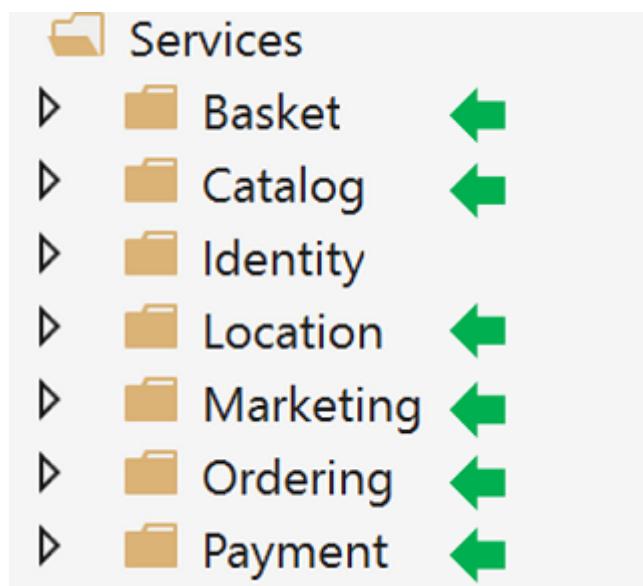


Figure 6-29. Microservice folders in eShopOnContainers solution in Visual Studio

About the Identity service, in the design it's left out of the API Gateway routing because it's the only cross-cutting concern in the system, although with Ocelot it's also possible to include it as part of the rerouting lists.

All those services are currently implemented as ASP.NET Core Web API services, as you can tell from the code. Let's focus on one of the microservices like the Catalog microservice code.

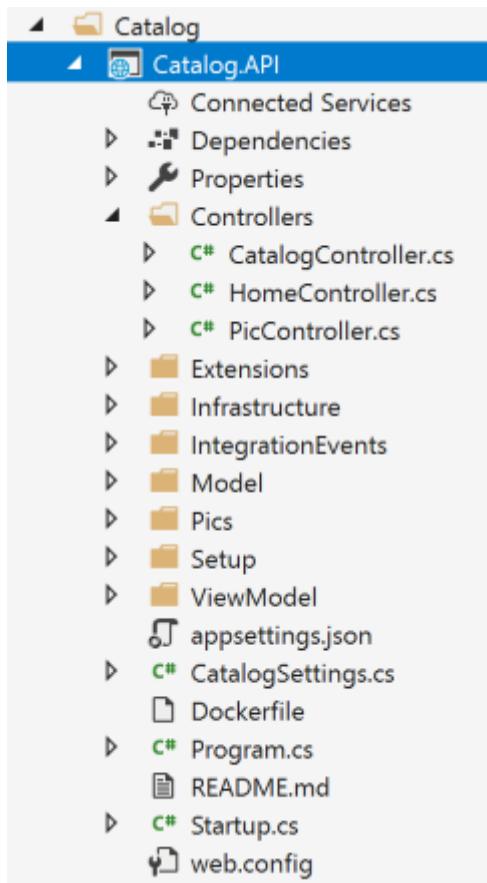


Figure 6-30. Sample Web API microservice (Catalog microservice)

You can see that the Catalog microservice is a typical ASP.NET Core Web API project with several controllers and methods like in the following code.

```
C#  
  
[HttpGet]  
[Route("items/{id:int}")]  
[ProducesResponseType((int) HttpStatusCode.BadRequest)]  
[ProducesResponseType((int) HttpStatusCode.NotFound)]  
[ProducesResponseType(typeof(CatalogItem), (int) HttpStatusCode.OK)]  
public async Task<IActionResult> GetItemById(int id)  
{  
    if (id <= 0)  
    {  
        return BadRequest();  
    }  
    var item = await _catalogContext.CatalogItems.  
        SingleOrDefaultAsync(ci => ci.Id  
== id);  
    //...  
  
    if (item != null)  
    {  
        return Ok(item);  
    }  
}
```

```
        return NotFound();
    }
```

The HTTP request will end up running that kind of C# code accessing the microservice database and any additional required action.

Regarding the microservice URL, when the containers are deployed in your local development PC (local Docker host), each microservice's container always has an internal port (usually port 80) specified in its dockerfile, as in the following dockerfile:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80
```

The port 80 shown in the code is internal within the Docker host, so it can't be reached by client apps.

Client apps can access only the external ports (if any) published when deploying with `docker-compose`.

Those external ports shouldn't be published when deploying to a production environment. For this specific reason, why you want to use the API Gateway, to avoid the direct communication between the client apps and the microservices.

However, when developing, you want to access the microservice/container directly and run it through Swagger. That's why in eShopOnContainers, the external ports are still specified even when they won't be used by the API Gateway or the client apps.

Here's an example of the `docker-compose.override.yml` file for the Catalog microservice:

yml

```
catalog-api:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:80
    - ConnectionString=YOUR_VALUE
    - ... Other Environment Variables
  ports:
    - "5101:80"  # Important: In a production environment you should remove
      # the external port (5101) kept here for microservice debugging purposes.
      # The API Gateway redirects and access through the
      # internal port (80).
```

You can see how in the `docker-compose.override.yml` configuration the internal port for the Catalog container is port 80, but the port for external access is 5101. But this port shouldn't be used by the application when using an API Gateway, only to debug, run, and test just the Catalog microservice.

Normally, you won't be deploying with `docker-compose` into a production environment because the right production deployment environment for microservices is an orchestrator like Kubernetes or Service Fabric. When deploying to those environments you use different configuration files where you won't publish directly any external port for the microservices but, you'll always use the reverse proxy from the API Gateway.

Run the catalog microservice in your local Docker host. Either run the full `eShopOnContainers` solution from Visual Studio (it runs all the services in the `docker-compose` files), or start the Catalog microservice with the following `docker-compose` command in CMD or PowerShell positioned at the folder where the `docker-compose.yml` and `docker-compose.override.yml` are placed.

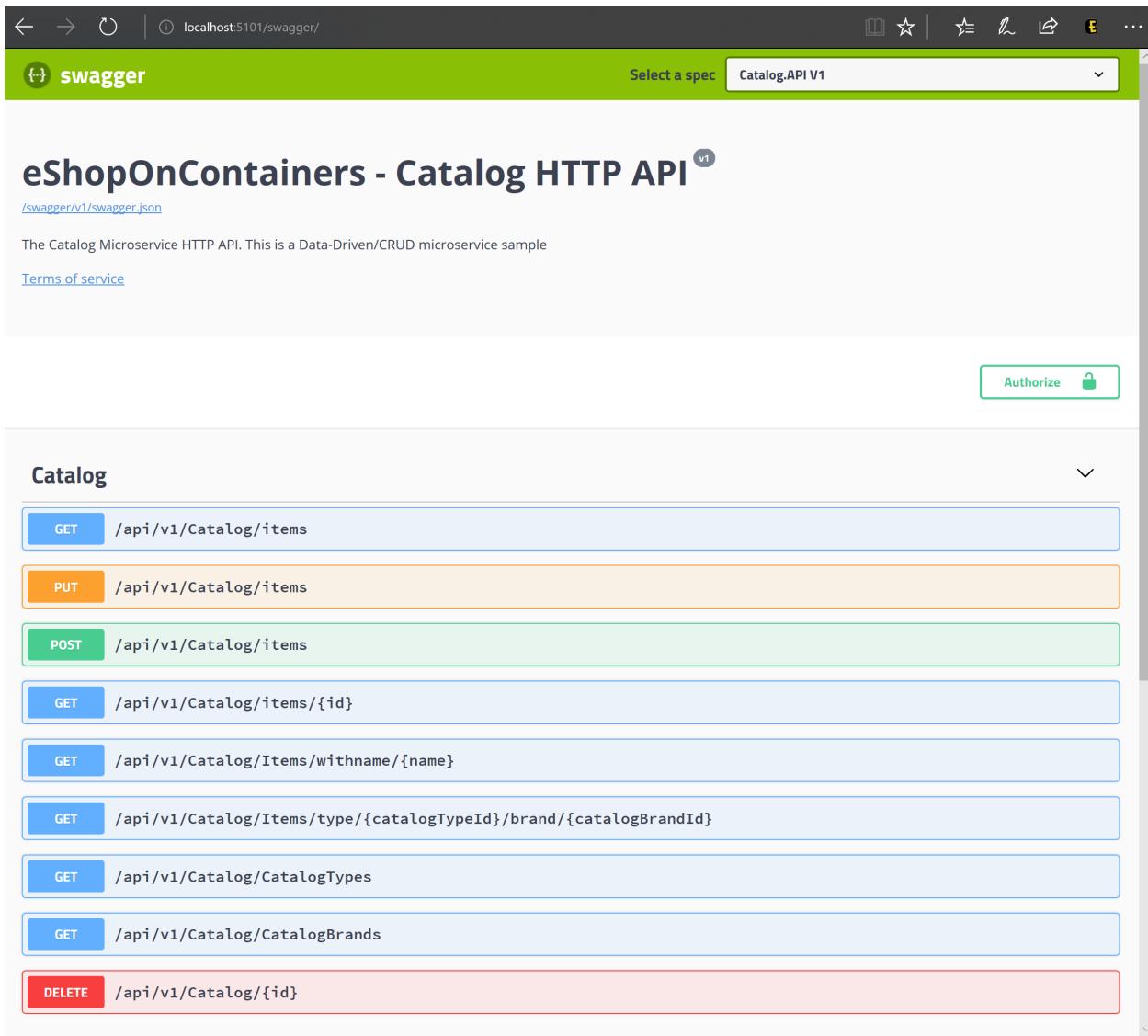
Console

```
docker-compose run --service-ports catalog-api
```

This command only runs the `catalog-api` service container plus dependencies that are specified in the `docker-compose.yml`. In this case, the SQL Server container and RabbitMQ container.

Then, you can directly access the Catalog microservice and see its methods through the Swagger UI accessing directly through that "external" port, in this case

`http://host.docker.internal:5101/swagger`:



The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample

[Terms of service](#)

[Authorize](#)

Catalog

- GET** /api/v1/Catalog/items
- PUT** /api/v1/Catalog/items
- POST** /api/v1/Catalog/items
- GET** /api/v1/Catalog/items/{id}
- GET** /api/v1/Catalog/Items/withname/{name}
- GET** /api/v1/Catalog/Items/type/{catalogTypeId}/brand/{catalogBrandId}
- GET** /api/v1/Catalog/CatalogTypes
- GET** /api/v1/Catalog/CatalogBrands
- DELETE** /api/v1/Catalog/{id}

Figure 6-31. Testing the Catalog microservice with its Swagger UI

At this point, you could set a breakpoint in C# code in Visual Studio, test the microservice with the methods exposed in Swagger UI, and finally clean-up everything with the `docker-compose down` command.

However, direct-access communication to the microservice, in this case through the external port 5101, is precisely what you want to avoid in your application. And you can avoid that by setting the additional level of indirection of the API Gateway (Ocelot, in this case). That way, the client app won't directly access the microservice.

Implementing your API Gateways with Ocelot

Ocelot is basically a set of middleware that you can apply in a specific order.

Ocelot is designed to work with ASP.NET Core only. The latest version of the package is 18.0 which targets .NET 6 and hence is not suitable for .NET Framework applications.

You install Ocelot and its dependencies in your ASP.NET Core project with [Ocelot's NuGet package](#), from Visual Studio.

```
PowerShell

Install-Package Ocelot
```

In eShopOnContainers, its API Gateway implementation is a simple ASP.NET Core WebHost project, and Ocelot's middleware handles all the API Gateway features, as shown in the following image:

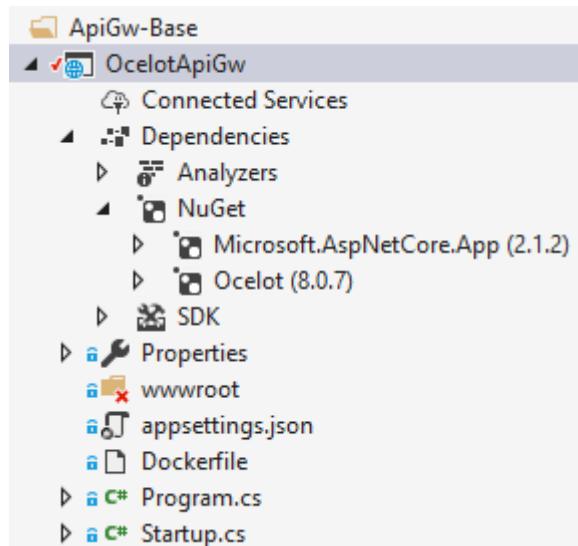


Figure 6-32. The OcelotApiGw base project in eShopOnContainers

This ASP.NET Core WebHost project is built with two simple files: `Program.cs` and `Startup.cs`.

The `Program.cs` just needs to create and configure the typical ASP.NET Core `BuildWebHost`.

```
C#
```

```
namespace OcelotApiGw
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args)
        {
            var builder = WebHost.CreateDefaultBuilder(args);
        }
    }
}
```

```

        builder.ConfigureServices(s => s.AddSingleton(builder))
            .ConfigureAppConfiguration(
                ic => ic.AddJsonFile(Path.Combine("configuration",
                    "configuration.json")))
            .UseStartup<Startup>();
        var host = builder.Build();
        return host;
    }
}
}

```

The important point here for Ocelot is the `configuration.json` file that you must provide to the builder through the `AddJsonFile()` method. That `configuration.json` is where you specify all the API Gateway ReRoutes, meaning the external endpoints with specific ports and the correlated internal endpoints, usually using different ports.

JSON

```
{
    "ReRoutes": [],
    "GlobalConfiguration": {}
}
```

There are two sections to the configuration. An array of ReRoutes and a GlobalConfiguration. The ReRoutes are the objects that tell Ocelot how to treat an upstream request. The Global configuration allows overrides of ReRoute specific settings. It's useful if you don't want to manage lots of ReRoute specific settings.

Here's a simplified example of [ReRoute configuration file](#) from one of the API Gateways from eShopOnContainers.

JSON

```
{
    "ReRoutes": [
        {
            "DownstreamPathTemplate": "/api/{version}/{everything}",
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "catalog-api",
                    "Port": 80
                }
            ],
            "UpstreamPathTemplate": "/api/{version}/c/{everything}",
            "UpstreamHttpMethod": [ "POST", "PUT", "GET" ]
        },
        {

```

```

    "DownstreamPathTemplate": "/api/{version}/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
        {
            "Host": "basket-api",
            "Port": 80
        }
    ],
    "UpstreamPathTemplate": "/api/{version}/b/{everything}",
    "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
    "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
    }
},
],
"GlobalConfiguration": {
    "RequestIdKey": "OcRequestId",
    "AdministrationPath": "/administration"
}
}

```

The main functionality of an Ocelot API Gateway is to take incoming HTTP requests and forward them on to a downstream service, currently as another HTTP request. Ocelot's describes the routing of one request to another as a ReRoute.

For instance, let's focus on one of the ReRoutes in the configuration.json from above, the configuration for the Basket microservice.

JSON

```
{
    "DownstreamPathTemplate": "/api/{version}/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
        {
            "Host": "basket-api",
            "Port": 80
        }
    ],
    "UpstreamPathTemplate": "/api/{version}/b/{everything}",
    "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
    "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
    }
}
```

The `DownstreamPathTemplate`, `Scheme`, and `DownstreamHostAndPorts` make the internal microservice URL that this request will be forwarded to.

The port is the internal port used by the service. When using containers, the port specified at its dockerfile.

The `Host` is a service name that depends on the service name resolution you are using. When using docker-compose, the services names are provided by the Docker Host, which is using the service names provided in the docker-compose files. If using an orchestrator like Kubernetes or Service Fabric, that name should be resolved by the DNS or name resolution provided by each orchestrator.

`DownstreamHostAndPorts` is an array that contains the host and port of any downstream services that you wish to forward requests to. Usually this configuration will just contain one entry but sometimes you might want to load balance requests to your downstream services and Ocelot lets you add more than one entry and then select a load balancer. But if using Azure and any orchestrator it is probably a better idea to load balance with the cloud and orchestrator infrastructure.

The `UpstreamPathTemplate` is the URL that Ocelot will use to identify which `DownstreamPathTemplate` to use for a given request from the client. Finally, the `UpstreamHttpMethod` is used so Ocelot can distinguish between different requests (GET, POST, PUT) to the same URL.

At this point, you could have a single Ocelot API Gateway (ASP.NET Core WebHost) using one or [multiple merged configuration.json files](#) or you can also store the [configuration in a Consul KV store](#).

But as introduced in the architecture and design sections, if you really want to have autonomous microservices, it might be better to split that single monolithic API Gateway into multiple API Gateways and/or BFF (Backend for Frontend). For that purpose, let's see how to implement that approach with Docker containers.

Using a single Docker container image to run multiple different API Gateway / BFF container types

In eShopOnContainers, we're using a single Docker container image with the Ocelot API Gateway but then, at run time, we create different services/containers for each type of API-Gateway/BFF by providing a different configuration.json file, using a docker volume to access a different PC folder for each service.

Containers API Gateways / BFF

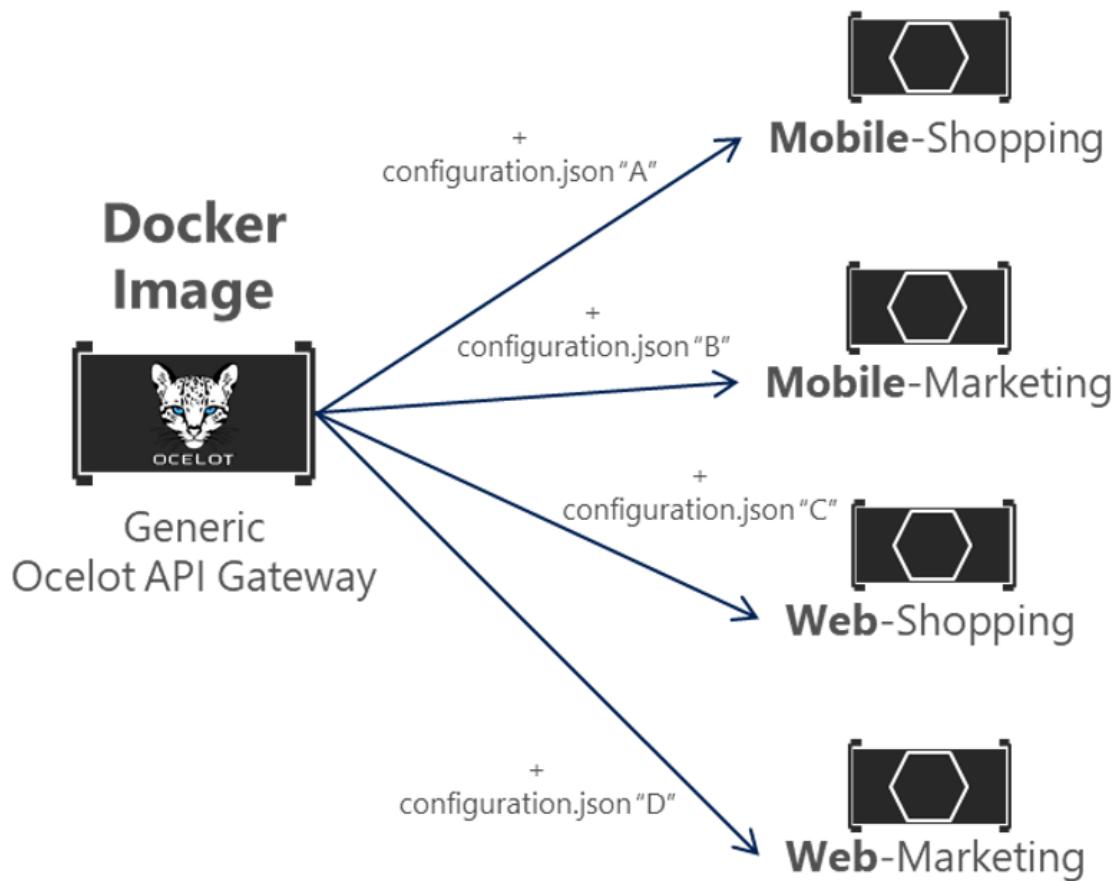


Figure 6-33. Reusing a single Ocelot Docker image across multiple API Gateway types

In eShopOnContainers, the "Generic Ocelot API Gateway Docker Image" is created with the project named 'OcelotApiGw' and the image name "eshop/ocelotapigw" that is specified in the docker-compose.yml file. Then, when deploying to Docker, there will be four API-Gateway containers created from that same Docker image, as shown in the following extract from the docker-compose.yml file.

```
yaml

mobileshoppingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

mobilemarketingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile
```

```

webshoppingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

webmarketingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

```

Additionally, as you can see in the following docker-compose.override.yml file, the only difference between those API Gateway containers is the Ocelot configuration file, which is different for each service container and it's specified at run time through a Docker volume.

yml

```

mobileshoppingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5200:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Shopping/apigw:/app/configuration

mobilemarketingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5201:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Marketeting/apigw:/app/configuration

webshoppingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5202:80"
  volumes:
    - ./src/ApiGateways/Web.Bff.Shopping/apigw:/app/configuration

webmarketingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:

```

```
- "5203:80"
volumes:
- ./src/ApiGateways/Web.Bff.Marketeting/apigw:/app/configuration
```

Because of that previous code, and as shown in the Visual Studio Explorer below, the only file needed to define each specific business/BFF API Gateway is just a configuration.json file, because the four API Gateways are based on the same Docker image.

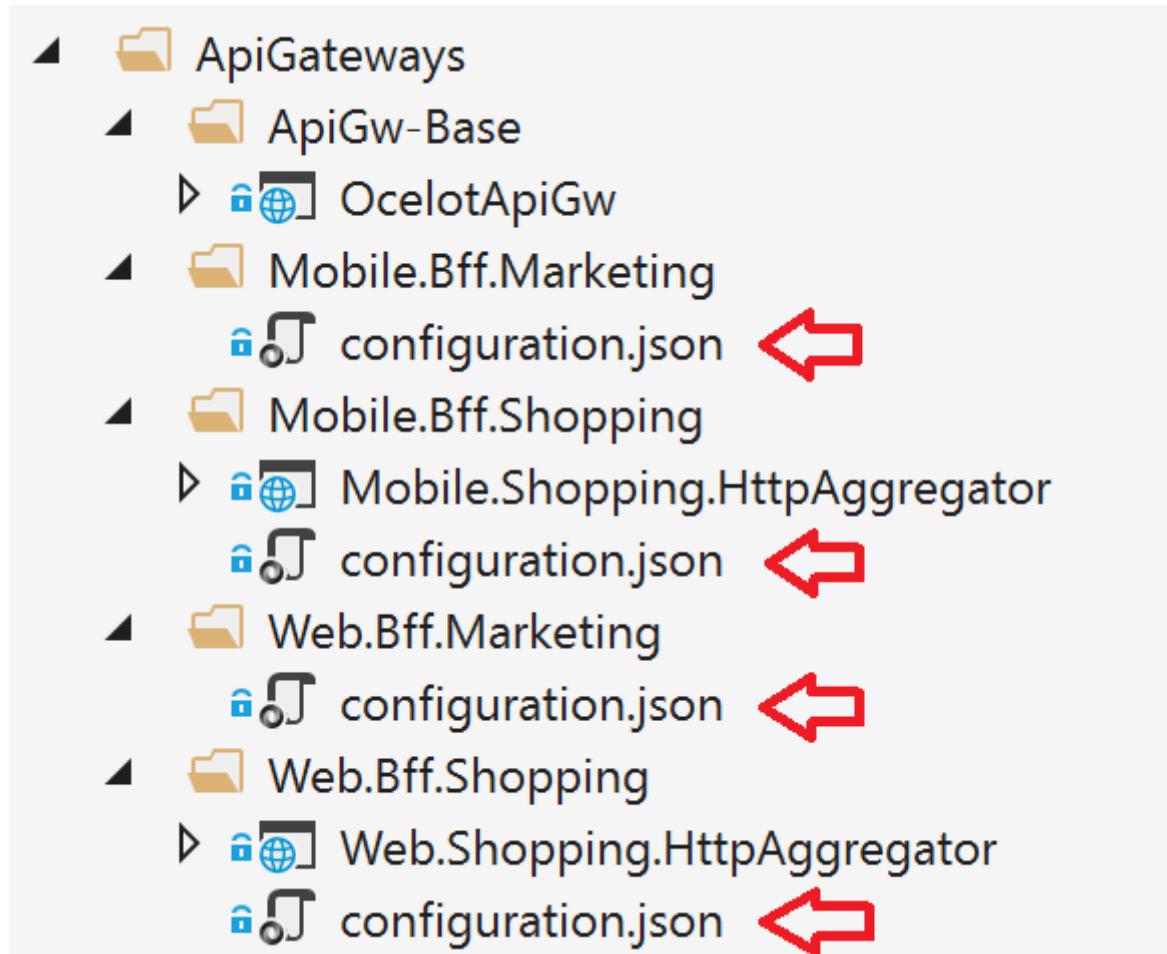


Figure 6-34. The only file needed to define each API Gateway / BFF with Ocelot is a configuration file

By splitting the API Gateway into multiple API Gateways, different development teams focusing on different subsets of microservices can manage their own API Gateways by using independent Ocelot configuration files. Plus, at the same time they can reuse the same Ocelot Docker image.

Now, if you run eShopOnContainers with the API Gateways (included by default in VS when opening eShopOnContainers-ServicesAndWebApps.sln solution or if running "docker-compose up"), the following sample routes will be performed.

For instance, when visiting the upstream URL

<http://host.docker.internal:5202/api/v1/c/catalog/items/2/> served by the

webshoppingapigw API Gateway, you get the same result from the internal Downstream URL `http://catalog-api/api/v1/2` within the Docker host, as in the following browser.

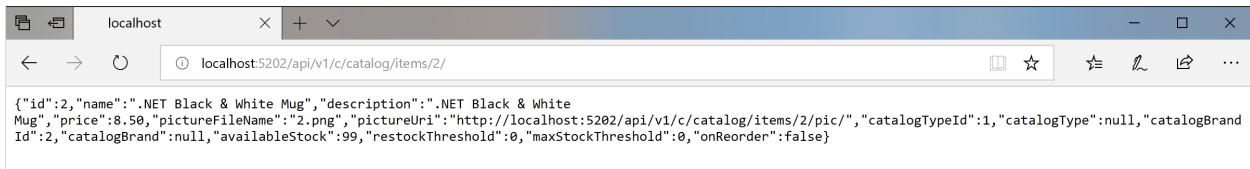


Figure 6-35. Accessing a microservice through a URL provided by the API Gateway

Because of testing or debugging reasons, if you wanted to directly access to the Catalog Docker container (only at the development environment) without passing through the API Gateway, since 'catalog-api' is a DNS resolution internal to the Docker host (service discovery handled by docker-compose service names), the only way to directly access the container is through the external port published in the docker-compose.override.yml, which is provided only for development tests, such as

`http://host.docker.internal:5101/api/v1/Catalog/items/1` in the following browser.

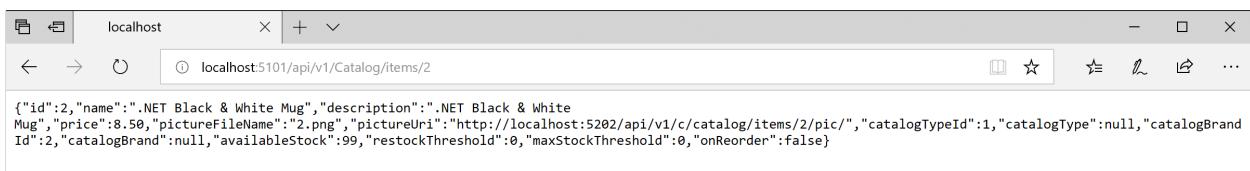


Figure 6-36. Direct access to a microservice for testing purposes

But the application is configured so it accesses all the microservices through the API Gateways, not through the direct port "shortcuts".

The Gateway aggregation pattern in eShopOnContainers

As introduced previously, a flexible way to implement requests aggregation is with custom services, by code. You could also implement request aggregation with the [Request Aggregation feature in Ocelot](#), but it might not be as flexible as you need. Therefore, the selected way to implement aggregation in eShopOnContainers is with an explicit ASP.NET Core Web API service for each aggregator.

According to that approach, the API Gateway composition diagram is in reality a bit more extended when considering the aggregator services that are not shown in the simplified global architecture diagram shown previously.

In the following diagram, you can also see how the aggregator services work with their related API Gateways.

eShopOnContainers

(API Gateways / BFF and Aggregator-services details)

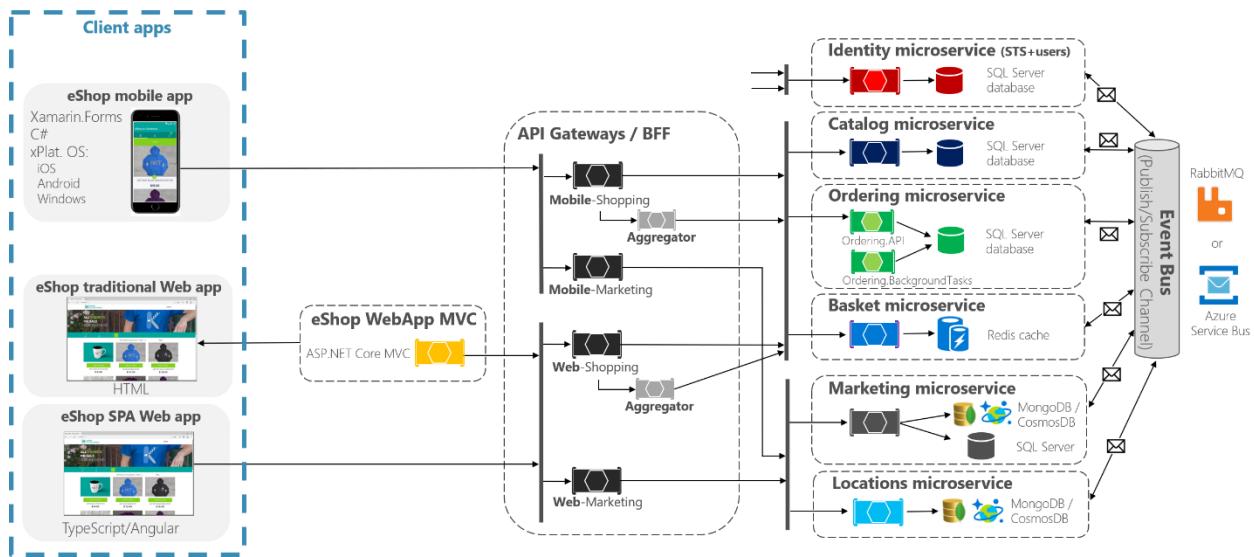


Figure 6-37. eShopOnContainers architecture with aggregator services

Zooming in further, on the "Shopping" business area in the following image, you can see that chattiness between the client apps and the microservices is reduced when using the aggregator services in the API Gateways.

eShopOnContainers

(API Gateways / BFF and Aggregator-services zoom-in)

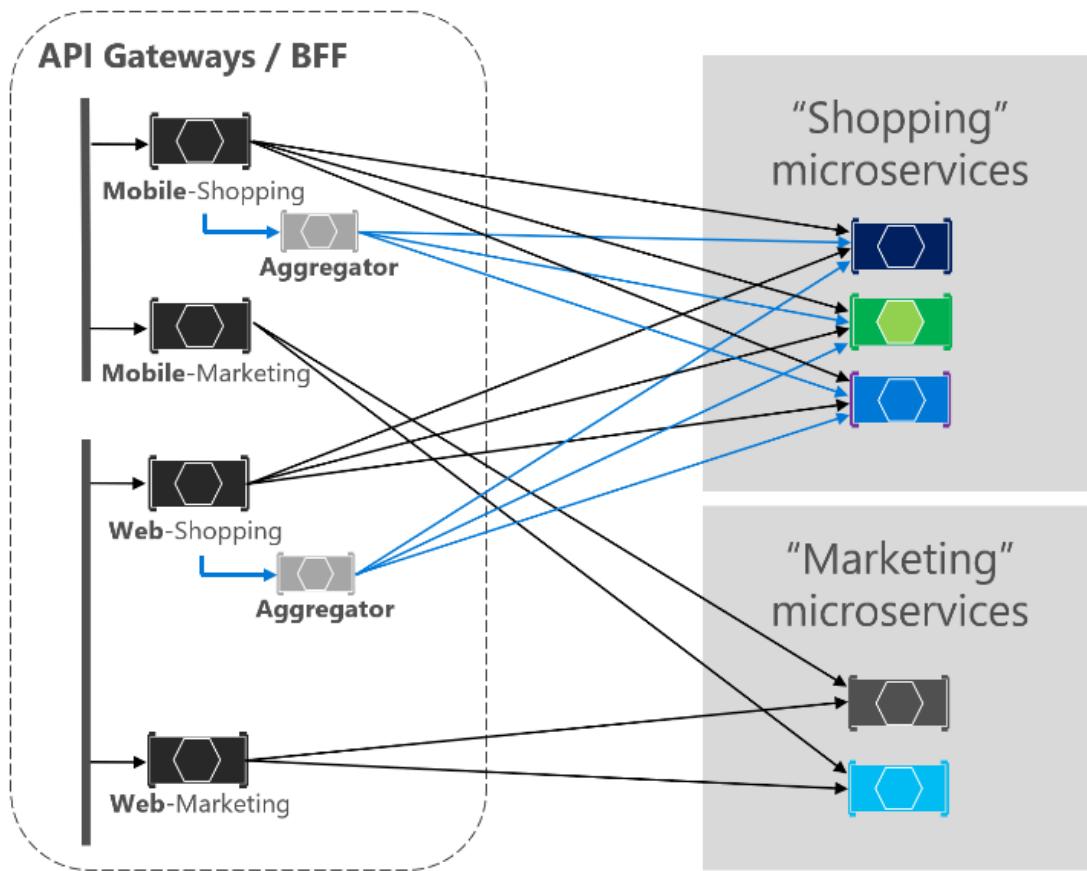


Figure 6-38. Zoom in vision of the Aggregator services

You can notice how when the diagram shows the possible requests coming from the API Gateways it can get complex. On the other hand, when you use the aggregator pattern, you can see how the arrows in blue would simplify the communication from a client app perspective. This pattern not only helps to reduce the chattiness and latency in the communication, it also improves the user experience significantly for the remote apps (mobile and SPA apps).

In the case of the "Marketing" business area and microservices, it is a simple use case so there was no need to use aggregators, but it could also be possible, if needed.

Authentication and authorization in Ocelot API Gateways

In an Ocelot API Gateway, you can sit the authentication service, such as an ASP.NET Core Web API service using [IdentityServer](#) providing the auth token, either out or inside the API Gateway.

Since eShopOnContainers is using multiple API Gateways with boundaries based on BFF and business areas, the Identity/Auth service is left out of the API Gateways, as highlighted in yellow in the following diagram.

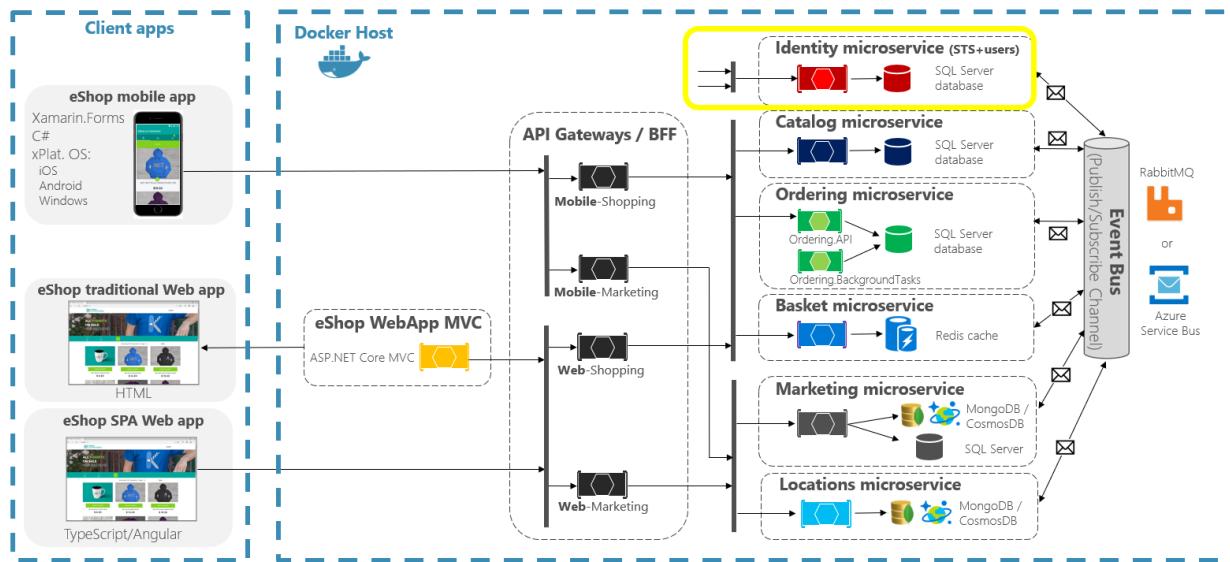


Figure 6-39. Position of the Identity service in eShopOnContainers

However, Ocelot also supports sitting the Identity/Auth microservice within the API Gateway boundary, as in this other diagram.

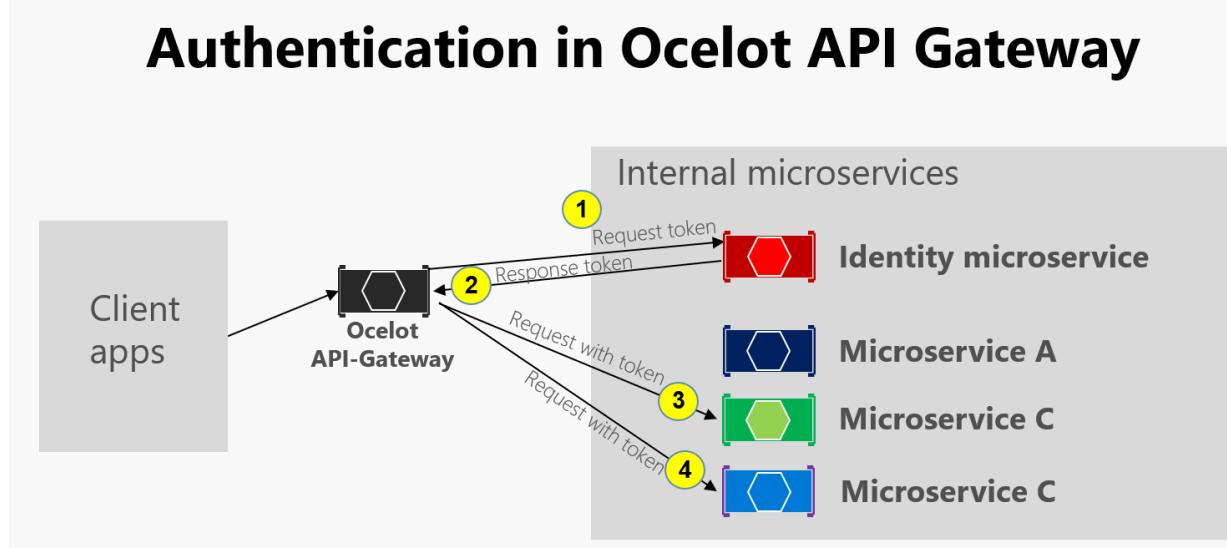


Figure 6-40. Authentication in Ocelot

As the previous diagram shows, when the Identity microservice is beneath the API gateway (AG): 1) AG requests an auth token from identity microservice, 2) The identity microservice returns token to AG, 3-4) AG requests from microservices using the auth token. Because eShopOnContainers application has split the API Gateway into multiple BFF (Backend for Frontend) and business areas API Gateways, another option would have been to create an additional API Gateway for cross-cutting concerns. That choice would be fair in a more complex microservice based architecture with multiple cross-

cutting concerns microservices. Since there's only one cross-cutting concern in eShopOnContainers, it was decided to just handle the security service out of the API Gateway realm, for simplicity's sake.

In any case, if the app is secured at the API Gateway level, the authentication module of the Ocelot API Gateway is visited at first when trying to use any secured microservice. That redirects the HTTP request to visit the Identity or auth microservice to get the access token so you can visit the protected services with the access_token.

The way you secure with authentication any service at the API Gateway level is by setting the AuthenticationProviderKey in its related settings at the configuration.json.

JSON

```
{  
  "DownstreamPathTemplate": "/api/{version}/{everything}",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "basket-api",  
      "Port": 80  
    }  
  ],  
  "UpstreamPathTemplate": "/api/{version}/b/{everything}",  
  "UpstreamHttpMethod": [],  
  "AuthenticationOptions": {  
    "AuthenticationProviderKey": "IdentityApiKey",  
    "AllowedScopes": []  
  }  
}
```

When Ocelot runs, it will look at the ReRoutes

AuthenticationOptions.AuthenticationProviderKey and check that there is an Authentication Provider registered with the given key. If there isn't, then Ocelot will not start up. If there is, then the ReRoute will use that provider when it executes.

Because the Ocelot WebHost is configured with the `authenticationProviderKey = "IdentityApiKey"`, that will require authentication whenever that service has any requests without any auth token.

C#

```
namespace OcelotApiGw  
{  
  public class Startup  
  {  
    private readonly IConfiguration _cfg;
```

```

        public Startup(IConfiguration configuration) => _cfg =
configuration;

        public void ConfigureServices(IServiceCollection services)
{
    var identityUrl = _cfg.GetValue<string>("IdentityUrl");
    var authenticationProviderKey = "IdentityApiKey";
    //...
    services.AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, x =>
    {
        x.Authority = identityUrl;
        x.RequireHttpsMetadata = false;
        x.TokenValidationParameters = new
Microsoft.IdentityModel.Tokens.TokenValidationParameters()
        {
            ValidAudiences = new[] { "orders", "basket",
"locations", "marketing", "mobileshoppingagg", "webshoppingagg" }
        };
    });
    //...
}
}
}

```

Then, you also need to set authorization with the `[Authorize]` attribute on any resource to be accessed like the microservices, such as in the following Basket microservice controller.

C#

```

namespace Microsoft.eShopOnContainers.Services.Basket.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class BasketController : Controller
    {
        //...
    }
}

```

The `ValidAudiences` such as "basket" are correlated with the audience defined in each microservice with `AddJwtBearer()` at the `ConfigureServices()` of the `Startup` class, such as in the code below.

C#

```

// prevent from mapping "sub" claim to nameidentifier.
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

```

```

var identityUrl = Configuration.GetValue<string>("IdentityUrl");

services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>
{
    options.Authority = identityUrl;
    options.RequireHttpsMetadata = false;
    options.Audience = "basket";
});

```

If you try to access any secured microservice, like the Basket microservice with a ReRoute URL based on the API Gateway like

`http://host.docker.internal:5202/api/v1/b/basket/1`, then you'll get a 401

Unauthorized unless you provide a valid token. On the other hand, if a ReRoute URL is authenticated, Ocelot will invoke whatever downstream scheme is associated with it (the internal microservice URL).

Authorization at Ocelot's ReRoutes tier. Ocelot supports claims-based authorization evaluated after the authentication. You set the authorization at a route level by adding the following lines to the ReRoute configuration.

JSON

```

"RouteClaimsRequirement": {
    "UserType": "employee"
}

```

In that example, when the authorization middleware is called, Ocelot will find if the user has the claim type 'UserType' in the token and if the value of that claim is 'employee'. If it isn't, then the user will not be authorized and the response will be 403 forbidden.

Using Kubernetes Ingress plus Ocelot API Gateways

When using Kubernetes (like in an Azure Kubernetes Service cluster), you usually unify all the HTTP requests through the [Kubernetes Ingress tier](#) based on *Nginx*.

In Kubernetes, if you don't use any ingress approach, then your services and pods have IPs only routable by the cluster network.

But if you use an ingress approach, you'll have a middle tier between the Internet and your services (including your API Gateways), acting as a reverse proxy.

As a definition, an Ingress is a collection of rules that allow inbound connections to reach the cluster services. An ingress is configured to provide services externally reachable URLs, load balance traffic, SSL termination and more. Users request ingress by POSTing the Ingress resource to the API server.

In eShopOnContainers, when developing locally and using just your development machine as the Docker host, you are not using any ingress but only the multiple API Gateways.

However, when targeting a "production" environment based on Kubernetes, eShopOnContainers is using an ingress in front of the API gateways. That way, the clients still call the same base URL but the requests are routed to multiple API Gateways or BFF.

API Gateways are front-ends or façades surfacing only the services but not the web applications that are usually out of their scope. In addition, the API Gateways might hide certain internal microservices.

The ingress, however, is just redirecting HTTP requests but not trying to hide any microservice or web app.

Having an ingress Nginx tier in Kubernetes in front of the web applications plus the several Ocelot API Gateways / BFF is the ideal architecture, as shown in the following diagram.

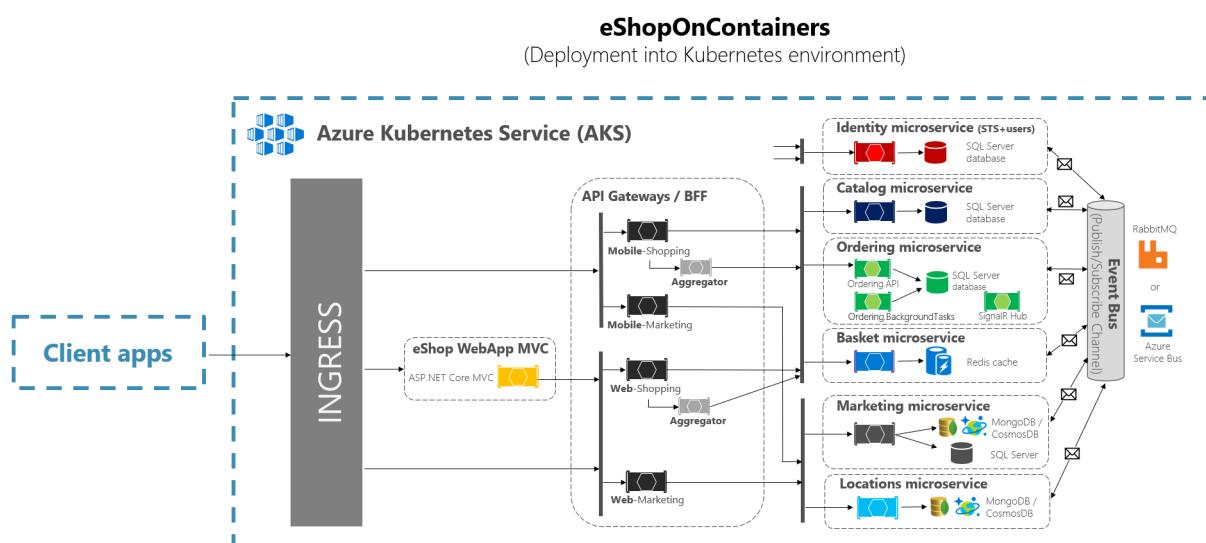


Figure 6-41. The ingress tier in eShopOnContainers when deployed into Kubernetes

A Kubernetes Ingress acts as a reverse proxy for all traffic to the app, including the web applications, that are out of the Api gateway scope. When you deploy eShopOnContainers into Kubernetes, it exposes just a few services or endpoints via *ingress*, basically the following list of postfixes on the URLs:

- `/` for the client SPA web application
- `/webmvc` for the client MVC web application
- `/webstatus` for the client web app showing the status/healthchecks
- `/webshoppingapigw` for the web BFF and shopping business processes
- `/webmarketingapigw` for the web BFF and marketing business processes
- `/mobileshoppingapigw` for the mobile BFF and shopping business processes
- `/mobilemarketingapigw` for the mobile BFF and marketing business processes

When deploying to Kubernetes, each Ocelot API Gateway is using a different "configuration.json" file for each *pod* running the API Gateways. Those "configuration.json" files are provided by mounting (originally with the `deploy.ps1` script) a volume created based on a Kubernetes *config map* named 'ocelot'. Each container mounts its related configuration file in the container's folder named `/app/configuration`.

In the source code files of eShopOnContainers, the original "configuration.json" files can be found within the `k8s/ocelot/` folder. There's one file for each BFF/APIGateway.

Additional cross-cutting features in an Ocelot API Gateway

There are other important features to research and use, when using an Ocelot API Gateway, described in the following links.

- **Service discovery in the client side integrating Ocelot with Consul or Eureka**
[https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html ↗](https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html)
- **Caching at the API Gateway tier**
[https://ocelot.readthedocs.io/en/latest/features/caching.html ↗](https://ocelot.readthedocs.io/en/latest/features/caching.html)
- **Logging at the API Gateway tier**
[https://ocelot.readthedocs.io/en/latest/features/logging.html ↗](https://ocelot.readthedocs.io/en/latest/features/logging.html)
- **Quality of Service (Retries and Circuit breakers) at the API Gateway tier**
[https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html ↗](https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html)

- **Rate limiting**

[https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html ↗](https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html)

- **Swagger for Ocelot**

[https://github.com/Burgyn/MMLib.SwaggerForOcelot ↗](https://github.com/Burgyn/MMLib.SwaggerForOcelot)

[Previous](#)

[Next](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Using Service Fabric to decompose applications

Azure Service Fabric

Azure Virtual Machines

In this example scenario, we walk through an approach using [Azure Service Fabric](#) as a platform for decomposing an unwieldy monolithic application. Here we consider an iterative approach to decomposing an IIS/ASP.NET web site into an application composed of multiple manageable microservices.

Moving from a monolithic architecture to a microservice architecture provides the following benefits:

- You can change one small, understandable unit of code and deploy only that unit.
- Each code unit requires just a few minutes or less to deploy.
- If there is an error in that small unit, only that unit stops working, not the whole application.
- Small units of code can be distributed easily and discretely among multiple development teams.
- New developers can quickly and easily grasp the discrete functionality of each unit.

A large IIS application on a server farm is used in this example, but the concepts of iterative decomposition and hosting can be used for any type of large application. While this solution uses Windows, Service Fabric can also run on Linux. It can be run on-premises, in Azure, or on VM nodes in the cloud provider of your choice.

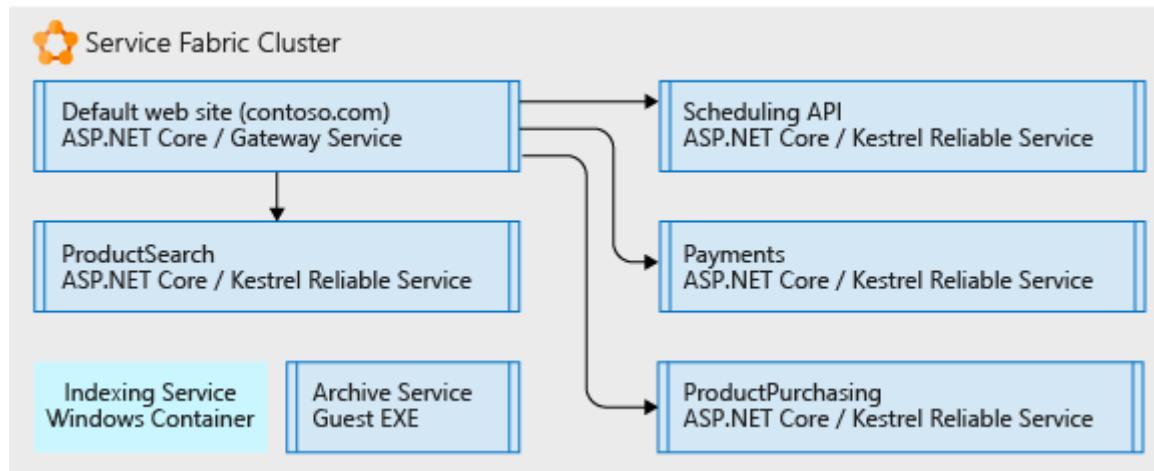
Potential use cases

This scenario is relevant to organizations with large monolithic Web applications that are experiencing:

- Errors in small code changes that break the entire website.
- Releases taking multiple days due to the need to release update the entire website.
- Long ramp-up times when onboarding new developers or teams due to the complex code base, requiring a single individual to know more than is feasible.

Architecture

Using Service Fabric as the hosting platform, we can convert a large IIS web site into a collection of microservices as shown below:



In the picture above, we decomposed all the parts of a large IIS application into:

- A routing or gateway service that accepts incoming browser requests, parses them to determine what service should handle them, and forwards the request to that service.
- Four ASP.NET Core applications that were formally virtual directories under the single IIS site running as ASP.NET applications. The applications were separated into their own independent microservices. The effect is that they can be changed, versioned, and upgraded separately. In this example, we rewrote each application using .NET Core and ASP.NET Core. These were written as **Reliable Services** so they can natively access the full Service Fabric platform capabilities and benefits (communication services, health reports, notifications, etc.).
- A Windows service called *Indexing Service*, placed in a Windows container so that it no longer makes direct changes to registry of the underlying server, but can run self-contained and be deployed with all its dependencies as a single unit.
- An Archive service, which is just an executable that runs according to a schedule and performs some tasks for the sites. It is hosted directly as a stand-alone executable because we determined it does what it needs to do without modification and it is not worth the investment to change.

Components

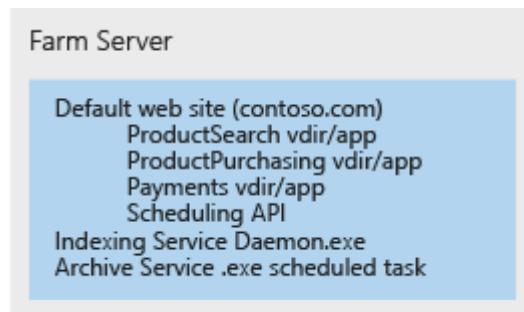
- [Service Fabric](#) is an open source project that you can use to build and operate always-on, scalable, distributed apps.

Considerations

The first challenge is to begin to identify smaller bits of code that can be factored out from the monolith into microservices that the monolith can call. Iteratively over time, the monolith is broken up into a collection of these microservices that developers can easily understand, change, and quickly deploy at low risk.

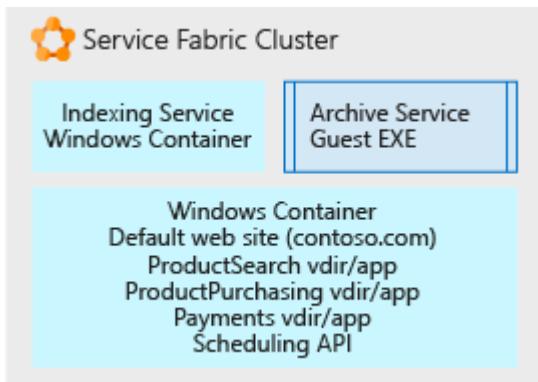
Service Fabric was chosen because it is capable of supporting running all the microservices in their various forms. For example you may have a mix of stand-alone executables, new small web sites, new small APIs, and containerized services, etc. Service Fabric can combine all these service types onto a single cluster.

To get to this final, decomposed application, we used an iterative approach. We started with a large IIS/ASP.NET web site on a server farm. A single node of the server farm is pictured below. It contains the original web site with several virtual directories, an additional Windows Service the site calls, and an executable that does some periodic site archive maintenance.



On the first development iteration, the IIS site and its virtual directories placed in a [Windows Container](#). Doing this allows the site to remain operational, but not tightly bound to the underlying server node OS. The container is run and orchestrated by the underlying Service Fabric node, but the node does not have to have any state that the site is dependent on (registry entries, files, etc.). All of those items are in the container. We have also placed the Indexing service in a Windows Container for the same reasons. The containers can be deployed, versioned, and scaled independently. Finally, we hosted the Archive Service a simple [stand-alone executable file](#) since it is a self-contained .exe with no special requirements.

The picture below shows how our large web site is now partially decomposed into independent units and ready to be decomposed more as time allows.



Further development focuses on separating the single large Default Web site container pictured above. Each of the virtual directory ASP.NET apps is removed from the container one at a time and ported to ASP.NET Core [Reliable Services](#).

Once each of the virtual directories has been factored out, the Default Web site is written as an ASP.NET Core reliable service, which accepts incoming browser requests and routes them to the correct ASP.NET application.

Availability, scalability, and security

Service Fabric is [capable of supporting various forms of microservices](#) while keeping calls between them on the same cluster fast and simple. Service Fabric is a [fault tolerant](#), self-healing cluster that can run containers, executables, and even has a native API for writing microservices directly to it (the 'Reliable Services' referred to above). The platform facilitates rolling upgrades and versioning of each microservice. You can tell the platform to run more or fewer of any given microservice distributed across the Service Fabric cluster in order to [scale](#) in or out only the microservices you need.

Service Fabric is a cluster built on an infrastructure of virtual (or physical) nodes, which have networking, storage, and an operating system. As such, it has a set of administrative, maintenance, and monitoring tasks.

You'll also want to consider governance and control of the cluster. Just as you would not want people arbitrarily deploying databases to your production database server, neither would you want people deploying applications to the Service Fabric cluster without some oversight.

Service Fabric is capable of hosting many different [application scenarios](#), take some time to see which ones apply to your scenario.

Pricing

For a Service Fabric cluster hosted in Azure, the largest part of the cost is the number and size of the nodes in your cluster. Azure allows quick and simple creation of a cluster composed of the underlying node size you specify, but the compute charges are based on the node size multiplied by the number of nodes.

Other less costly components of cost are the storage charges for each node's virtual disks and network I/O egress charges from Azure (for example network traffic out of Azure to a user's browser).

To get an idea of cost, we have created an example using some default values for cluster size, networking, and storage: Take a look at the [pricing calculator](#). Feel free to update the values in this default calculator to those relevant to your situation.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Tim Omta](#) | Senior Cloud Solution Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Take some time to familiarize yourself with the platform by going through the [documentation](#) and reviewing the many different [application scenarios](#) for Service Fabric. The documentation will tell you what a cluster consists of, what it can run on, software architecture, and maintenance for it.

To see a demonstration of Service Fabric for an existing .NET application, deploy the Service Fabric [quickstart](#).

Here are some additional articles about Service Fabric:

- [Service Fabric overview](#)
- [Service Fabric programming model](#)
- [Service Fabric availability](#)
- [Scaling Service Fabric](#)
- [Hosting containers in Service Fabric](#)
- [Hosting standalone executables in Service Fabric](#)
- [Service Fabric Reliable Services](#)

- [Service Fabric application scenarios](#)

From the standpoint of your current application, start to think about its different functions. Choose one of them and think through how you can separate only that function from the whole. Take it one discrete, understandable, piece at a time.

Related resources

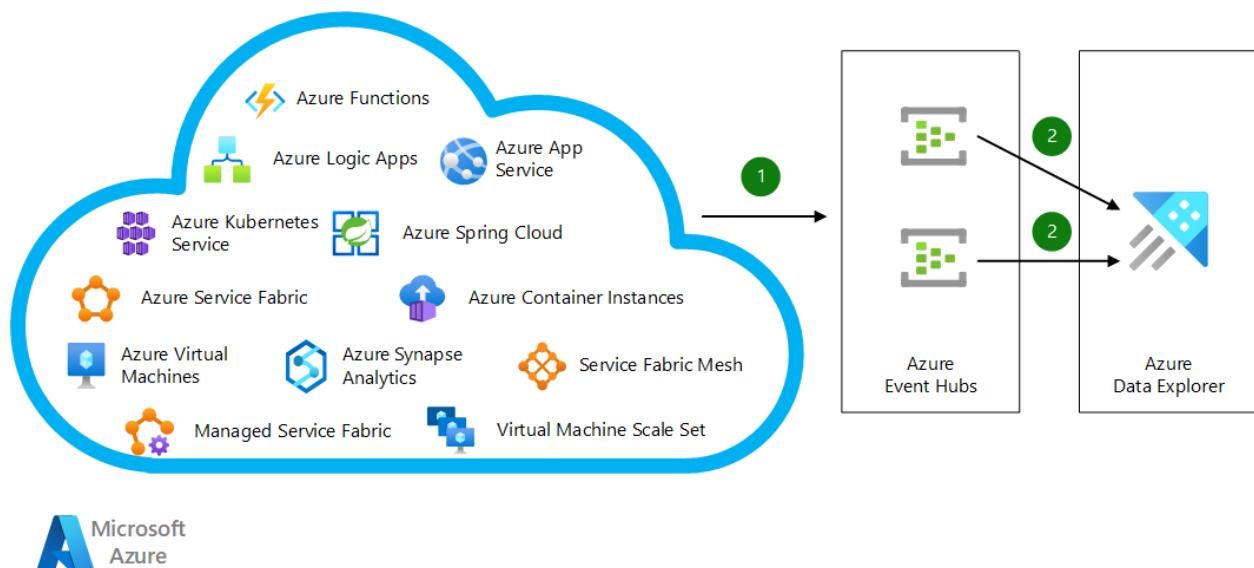
- [Building microservices on Azure](#)
- [Microservices assessment and readiness](#)
- [Design a microservices architecture](#)
- [Data considerations for microservices](#)

Enterprise-grade logging on Azure

Azure Data Explorer Azure Event Hubs Azure Functions

This reference architecture describes how to achieve enterprise-grade logging on Azure with a common logging method that enables end-to-end traceability across different applications. You can use these logs for troubleshooting and for finding insights in usage metrics and business events. This reference architecture is not a replacement for Azure Monitor. This architecture is mainly targeted for application logs. Infrastructure logs can be achieved with the Log Analytics component of Azure Monitor. Application performance monitoring can be achieved with the Application Insights component for Azure Monitor. [Deploy this scenario](#).

Architecture



Download a [Visio file](#) that contains this architecture diagram.

Workflow

1. Regardless of underlying technology or programming language, applications running on Azure compute services send their logs to shared or dedicated Azure Event Hubs instances by using built-in integration capabilities or the Event Hubs SDK.

2. Azure Data Explorer clusters subscribe to these event hubs and ingest data directly to the relevant databases. For high availability, you can define another consumer group in Event Hubs and create a new Azure Data Explorer cluster in another Azure region for dual ingestion.

Components

- [Event Hubs ↗](#)
- [Azure Data Explorer ↗](#)
- [Azure Event Hubs data connection to Azure Data Explorer](#)

Alternatives

- The Application Insights and Log Analytics features of [Azure Monitor](#) can do end-to-end tracing and troubleshooting. These features use Azure Data Explorer behind the scenes. Azure Monitor has capabilities and benefits that are similar to the current architecture, but it's a challenge for Azure Monitor to consolidate multiple applications into a single workspace. [Workspace-based Application Insights](#) provides a capability to centralize logs from Application Insights into a common Log Analytics workspace. Azure Monitor allows you to ingest custom logs and extend the data format for these custom logs. There are two different tradeoffs with this alternative:
 - Custom logs will be ingested to different tables than the original tables. `trackEvent()` for custom logs will be stored in the `customEvents` table, whereas the original events will be stored in the `Events` table.
 - Developers can extend table schema for custom tables as needed, which is a feature of Azure Monitor. Without a strict governance model, the table schema can become complicated over time. With the guidance from this architecture, you can introduce your own governance model and still benefit from [Workspace-based Application Insights](#) capabilities by running cross-cluster queries.
- [Microsoft Sentinel](#) can provide similar capabilities from a security standpoint, but it isn't suitable for application troubleshooting or end-to-end application traceability.

Scenario details

Applications use different tools and technologies that have their own formats and user interfaces to record errors, events, and traces. Merging this log data from different applications is a programming challenge. The challenge becomes more complicated

with distributed systems in the cloud and in high-scale environments. Logs from different systems cause similar problems for big data solutions.

For end-to-end traceability, it's important to design tables that share certain columns. Common columns and their data types include:

- `Timestamp` (datetime): The exact time the log is generated. Always set the time at origin, because there could be an ingestion delay, and ingestion order isn't guaranteed. Always log with UTC time to prevent any impact from time changes and time zone differences.
- `ActivityId` (string): A single unique activity identifier that can pass across different solution components. For example, a microservices single execution can include multiple calls to different services. Every table includes this identifier.
- `Tag` (string): An identifier for types of logs in a table. You can use GUIDs as tags. This tag should appear in source code to make it easier to distinguish records for the same activity.
- `Level` (int): Identifies the logging level of the record. The logging level for a particular component can change during a troubleshooting session. You can use this column to filter records to be ingested or aged differently. A common practice is using `0` for Trace, `1` for Debug, `2` for Information, `3` for Warning, `4` for Error, and `5` for Critical.
- `Properties` (dynamic): A bag that contains JSON-formatted key-value pairs. You can add computer name, IP address, process ID, operating system version, and so on.
- `DataVersion` (string): Identifies data formatting changes in the log entries. If you build your own dashboard that relies on logging data, you can use this field to handle breaking changes.

You can also add columns like `BinaryVersion` to identify your application version. Use this field to catch whether any of your running application instances are still using older versions.

You can include the following recommended reference tables in your logging design:

Events

This table contains events that occur during an execution flow. These events can be business events such as users signing in, or system events like processes starting.

[\[\] Expand table](#)

Column	Datatype	Remarks
Timestamp	datetime	Time the event happened
ActivityId	string	Unique identifier for end-to-end traceability
Tag	string	Type of event
Level	int	Logging level, such as Information or Warning
Properties	dynamic	Bag that contains relevant event properties
DataVersion	string	Data format version to identify formatting changes
Name	string	Human-readable description
Metrics	dynamic	Bag that contains metrics for the event in key-value pairs format

Metrics

This table contains aggregated performance metrics or other numeric data to describe aspects of a system at a particular time. These metrics can contain the duration of a call to an external system, or the data received or processed in a particular time window.

[\[\] Expand table](#)

Column	Datatype	Remarks
Timestamp	datetime	Metric collection time
ActivityId	string	Unique identifier for end-to-end traceability
Tag	string	Type of metric

Column	Datatype	Remarks
Level	int	Logging level, such as Information or Warning
Properties	dynamic	Bag that contains relevant event properties
DataVersion	string	Data format version to identify formatting changes
Name	string	Human-readable description
Count	int	Number of measurements captured over the aggregation interval
Sum	real	Sum of all values captured over the aggregation interval
Min	real	Smallest value captured over the aggregation interval
Max	real	Largest value captured over the aggregation interval

Traces

This table contains logs that increase traceability. The table can contain any logs that you don't need to associate with an event or metric. This table can grow enormously over time if you don't use it carefully. The table design is the same as the `Events` table.

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

Column	Datatype	Remarks
Timestamp	datetime	Time the event happened
ActivityId	string	Unique identifier for end-to-end traceability
Tag	string	Type of event
Level	int	Logging level, such as Information or Warning

Column	Datatype	Remarks
Properties	dynamic	Bag that contains relevant event properties
DataVersion	string	Data format version to identify formatting changes
Name	string	Human-readable description
Metrics	dynamic	Bag that contains metrics for the event in key-value pairs format

Exceptions

This table contains first-chance exceptions that the application catches. This table can also log second-chance exceptions if the underlying platform supports them.

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

Column	Datatype	Remarks
Timestamp	datetime	Time the exception was caught
ActivityId	string	Unique identifier for end-to-end traceability
Tag	string	Type of event for the log
Level	int	Logging level, usually Error
Properties	dynamic	Bag that contains relevant information
DataVersion	string	Data format version to identify formatting changes
Name	string	Human-readable description
UserMessage	string	Exception message

Column	Datatype	Remarks
StackTrace	dynamic	Bag containing stack traces, including inner exceptions if any

Potential use cases

Typical uses for this architecture include:

- Enterprise-grade or large-scale applications.
- Telemetry monitoring for internet of things (IoT)-based applications.
- Business activity monitoring for applications with high numbers of transactions.

Recommendations

Include the preceding recommended reference tables in your logging design as starting points to customize as needed. Depending on your business requirements and application model, you might need different tables and columns. You can introduce tables such as `Audits`, `Requests`, or `Dependencies`.

- For a web-based application, log `Requests` details, such as URL, request size, response size, response code, and timing.
- For a microservices-based application, a `Dependencies` table can log interactions among different components and services, such as caller, callee, time taken, and request and response details.
- For a transaction-based application, add an `Audit` table to log transaction details like when, what, who, and how.

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

This scenario relies on the availability of the Event Hubs and Azure Data Explorer services. If there's an Azure Data Explorer outage, Event Hubs retains its data for the number of days configured in its retention period, which depends on your [chosen tier](#).

If there's an Event Hubs outage, you could lose data. To avoid data loss, choose the right Event Hubs tier for your needs. As a backup mechanism, you could insert collected log data into blob storage and ingest the blob storage data to a Kusto cluster. For more information, see [Ingest multi-lined JSON records](#).

[Azure availability zones](#) are unique physical locations within an Azure region that can help protect Azure Data Explorer compute clusters and data from partial region failure.

Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

It's important not to log any sensitive data. This solution doesn't offer any explicit capability to prevent sensitive data from being logged. Azure Data Explorer can implement row level security based on your business requirements. For more information, see [Row level security](#) and [Security in Azure Data Explorer](#).

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

The cost of running this scenario depends on the number of logs or records generated, data retention policies defined, and queries executed on Azure Data Explorer. This calculation excludes the cost of running your applications.

- Event Hubs pricing is based on the number of records generated, pricing tier, and the number of Event Hubs instances. See [Event Hubs pricing](#).

Event Hubs Standard tier allows automatic scaling of throughput units. For more information, see [Automatically scale up Azure Event Hubs throughput units](#).

Auto-inflate doesn't automatically scale down the number of throughput units when ingress or egress rates drop below the limits, but you can use a simple script to scale down.

- For Azure Data Explorer, use the [Azure Data Explorer Cost Estimator](#) to estimate costs. Azure Data Explorer supports autoscaling to help you control compute charges. For more information, see [Manage cluster horizontal scaling \(scale out\) in Azure Data Explorer](#).

Setting retention and caching policies affects storage costs. See [Create a table's retention and cache policies](#).

You can size two Azure Data Explorer clusters differently to achieve [active-active](#) architecture. You can use one cluster only for ingestion but not querying, and use the larger cluster for both ingestion and querying.

Operational excellence

Operational excellence covers the operations processes that deploy an application and keep it running in production. For more information, see [Overview of the operational excellence pillar](#).

Azure Data Explorer provides built-in mechanisms for business continuity and disaster recovery. You can choose different solutions based on Recovery Point Objective (RPO), Recovery Time Objective (RTO), effort, and cost. For more information, see [Azure Data Explorer business continuity and disaster recovery](#).

You can easily achieve active-active configurations with this architecture by using the dual ingestion approach.

Performance efficiency

Performance efficiency is the ability of your workload to scale in an efficient manner to meet the demands placed on it by users. For more information, see [Performance efficiency pillar overview](#).

Ingesting individual records into an Azure Data Explorer cluster can be expensive. With Event Hubs integration, Azure Data Explorer ingests data in bulk to optimize performance. Bulk operations are processed faster than batch operations.

Bulk ingestion could introduce ingestion latency in a low-use application scenario. Batching policies can control ingestion latency. For more information, see [IngestionBatching policy](#). You can trigger ingestion by number of items, data size, or timespan to ensure an acceptable latency. It's important to balance latency and performance. Lowering latency results in higher Azure Data Explorer resource consumption.

The Event Hubs SDK uses [AMQP protocol](#) to deliver events to Event Hubs. Reusing existing Event Hubs client connections reduces latency and promotes performance efficiency.

You can scale Event Hubs based on the number of throughput units and partitions set at creation time. Choosing the appropriate partition numbers is important for performance and scalability. You can use as many event hubs as you need, and group applications or sources into multiple event hubs. For prescriptive guidance on Event Hubs scalability, see [Scaling with Event Hubs](#).

You can scale Azure Data Explorer both [horizontally](#) and [vertically](#).

Deploy this scenario

To deploy this scenario, create strongly typed class definitions for table columns that you can use to collect and store metrics from multiple applications. Send the data to Event Hubs, and connect Azure Data Explorer to ingest the Event Hubs data. Use Kusto Query Language in Azure Data Explorer to query the data.

Prerequisites

- You need an active Azure account. If you don't have one, [create a free account](#).
- In the Azure portal, create an Azure Data Explorer cluster and database by following the steps in [Quickstart: Create an Azure Data Explorer cluster and database](#).

Create a table with common columns

In the **Query** window for your Azure Data Explorer database, run the following script to create an `Events` table with recommended columns:

```
Kusto

.create-merge table Events (Timestamp:datetime, Tag:string,
ActivityId:string, Level:int, Name:string, Metrics:dynamic,
Properties:dynamic, DataVersion:string)
```

For more information about creating tables, see [Create a table in Azure Data Explorer](#).

Send logs to Event Hubs

Create strongly typed class definitions for each table to ensure that different teams within your organization use the same format. The following C# code shows a sample definition for the `Events` table. You can create similar typed objects in any programming language. Data ingests to Azure Data Explorer in JSON format.

C#

```
public class LoggingEvent {
    public DateTime Timestamp { get; set; }
    public string ActivityId { get; set; }
    public string Tag { get; set; }
    public int Level { get; set; } // Logging level
    public string Name { get; set; }
    public Dictionary<string, double> Metrics { get; set; }
    public Dictionary<string, string> Properties { get; set; }
    public string DataVersion { get; set; }
}
```

You can introduce a library to make usage easier for developers. You just need to create one instance of the class and populate the necessary data. For example, you can use the following code snippet to populate data for code running on an Azure Functions app.

C#

```
LoggingEvent loggingEvent = new LoggingEvent()
{
    Timestamp = DateTime.UtcNow,
    ActivityId = activityId,
    Tag = "B1C01FCF-BA48-4923-B312-C45E5EA30506",
    Level = 2,
    Name = "An important business event happened",
    Metrics = new Dictionary<string, double>(),
    Properties = new Dictionary<string, string>(),
    DataVersion = "1.0"
};

loggingEvent.Metrics.Add("DBExecution", random.NextDouble() * 1000.0);
loggingEvent.Metrics.Add("OverallExecution",
1000.0 + random.NextDouble() * 100.0);
loggingEvent.Properties.Add("MachineName", System.Environment.MachineName);
loggingEvent.Properties.Add("ProcessId",
System.Environment.ProcessId.ToString());
loggingEvent.Properties.Add("ProcessPath", System.Environment.ProcessPath);
loggingEvent.Properties.Add("FunctionName", context.FunctionName);
loggingEvent.Properties.Add("OSVersion",
System.Environment.OSVersion.ToString());
```

This record stores multiple metrics in a single record. The record contains database execution time and overall execution time. Execution times are set to random values for

demonstration purposes.

You can send captured data to an event hub. For detailed .NET examples, see [Send events to Azure Event Hubs in .NET](#). The same documentation section contains articles about using other code languages to send events.

Ingest data from Event Hubs to Azure Data Explorer

Azure Data Explorer can ingest data from Event Hubs. To create the necessary data connection, see [Connect to the event hub](#). You can then use Kusto Query Language to query the data your application sends, and create dashboards to visualize the captured data.

By following a similar approach, you can collect your infrastructure logs into the same Azure Data Explorer cluster. For more information, see [Query exported data from Azure Monitor using Azure Data Explorer](#).

You can also query data directly from Azure Monitor by using cross clusters. For more information, see [Query data in Azure Monitor using Azure Data Explorer](#).

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Yunus Emre Alpozen](#) | Program Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [What is Event Hubs?](#)
- [What is Azure Data Explorer?](#)
- [Application profiling considerations for performance monitoring](#)
- [Kusto Query Language](#)
- [Visualize data with Azure Data Explorer](#)

Related resources

- [Unified logging for microservices apps](#)

- Big data analytics with Azure Data Explorer
- Azure Data Explorer monitoring
- Long term security log retention with Azure Data Explorer
- Real time analytics on big data architecture

Microservices architecture on Azure Service Fabric

Azure API Management

Azure Key Vault

Azure Monitor

Azure Pipelines

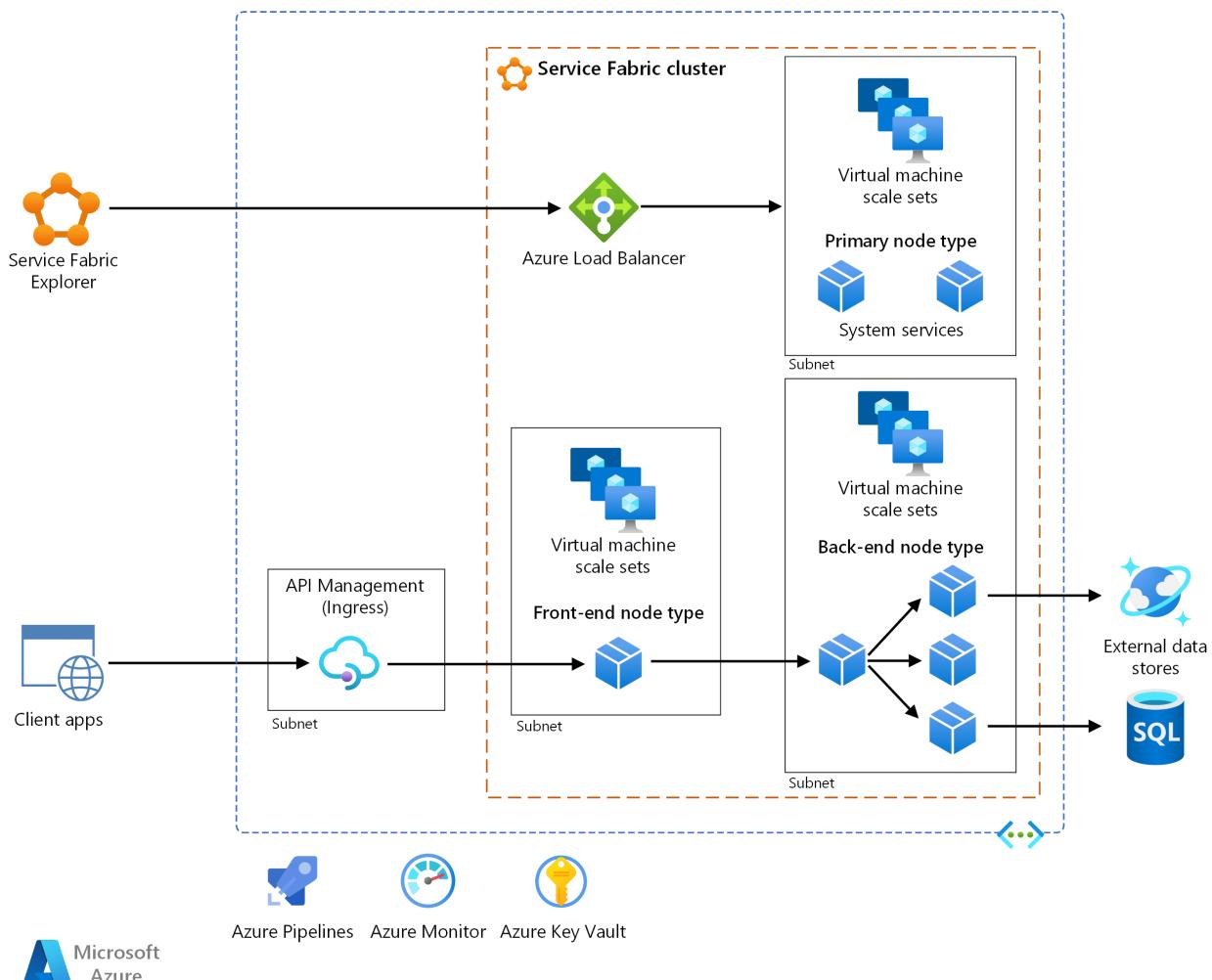
Azure Service Fabric

This reference architecture shows a microservices architecture deployed to Azure Service Fabric. It shows a basic cluster configuration that can be the starting point for most deployments.



A reference implementation of this architecture is available on [GitHub](#).

Architecture



Download a [Visio file](#) of this architecture.

! Note

This article focuses on the **Reliable Services** programming model for Service Fabric. Using Service Fabric to deploy and manage **containers** is beyond the scope of this article.

Workflow

The architecture consists of the following components. For other terms, see [Service Fabric terminology overview](#).

- **Service Fabric cluster.** A cluster is a network-connected set of virtual machines (VMs) into which you deploy and manage your microservices.
- **Virtual machine scale sets.** Virtual machine scale sets allow you to create and manage a group of identical, load-balanced, and autoscaling VMs. These compute resources also provide the fault and upgrade domains.
- **Nodes.** The nodes are the VMs that belong to the Service Fabric cluster.
- **Node types.** A node type represents a virtual machine scale set that deploys a collection of nodes. A Service Fabric cluster has at least one node type.

In a cluster that has multiple node types, one must be declared the [primary node type](#). The primary node type in the cluster runs the [Service Fabric system services](#). These services provide the platform capabilities of Service Fabric. The primary node type also acts as the [seed nodes](#), which are the nodes that maintain the availability of the underlying cluster.

Configure [additional node types](#) to run your services.

- **Services.** A service performs a standalone function that can start and run independently of other services. Instances of services get deployed to nodes in the cluster. There are two varieties of services in Service Fabric:
 - **Stateless service.** A stateless service does not maintain state within the service. If state persistence is required, then state is written to and retrieved from an external store, such as Azure Cosmos DB.
 - **Stateful service.** The [service state](#) is kept within the service itself. Most stateful services implement this through [Reliable Collections](#) in Service Fabric.
- **Service Fabric Explorer.** [Service Fabric Explorer](#) is an open-source tool for inspecting and managing Service Fabric clusters.
- **Azure Pipelines.** [Azure Pipelines](#) is part of [Azure DevOps Services](#) and runs automated builds, tests, and deployments. You can also use third-party continuous

integration and continuous delivery (CI/CD) solutions such as Jenkins.

- **Azure Monitor.** [Azure Monitor](#) collects and stores metrics and logs, including platform metrics for the Azure services in the solution and application telemetry. Use this data to monitor the application, set up alerts and dashboards, and perform root cause analysis of failures. Azure Monitor integrates with Service Fabric to collect metrics from controllers, nodes, and containers, along with container and node logs.
- **Azure Key Vault.** Use [Key Vault](#) to store any application secrets that the microservices use, such as connection strings.
- **Azure API Management.** In this architecture, [API Management](#) acts as an API gateway that accepts requests from clients and routes them to your services.

Considerations

These considerations implement the pillars of the [Azure Well-Architected Framework](#), which is a set of guiding tenets for improving the quality of a workload.

Design considerations

This reference architecture is focused on [microservices architectures](#). A microservice is a small, independently versioned unit of code. It's discoverable through service discovery mechanisms and can communicate with other services over APIs. Each service is self-contained and should implement a single business capability. For more information about how to decompose your application domain into microservices, see [Using domain analysis to model microservices](#).

Service Fabric provides an infrastructure to build, deploy, and upgrade microservices efficiently. It also provides options for autoscaling, managing state, monitoring health, and restarting services in case of failure.

Service Fabric follows an application model where an application is a collection of microservices. The application is described in an [application manifest](#) file. This file defines the types of services that the application contains, along with pointers to the independent service packages.

The application package also usually contains parameters that serve as overrides for certain settings that the services use. Each service package has a manifest file that describes the physical files and folders that are necessary to run that service, including

binaries, configuration files, and read-only data. Services and applications are independently versioned and upgradable.

Optionally, the application manifest can describe services that are automatically provisioned when an instance of the application is created. These are called *default services*. In this case, the application manifest also describes how these services should be created. That information includes the service's name, instance count, security or isolation policy, and placement constraints.

Note

Avoid using default services if you want to control the lifetime of your services. Default services are created when the application is created, and they run as long as the application is running.

For more information, see [So you want to learn about Service Fabric?](#).

Application-to-service packaging model

A tenet of microservices is that each service can be independently deployed. In Service Fabric, if you group all of your services into a single application package, and one service fails to be upgraded, the entire application upgrade gets rolled back. That rollback prevents other service from being upgraded.

For that reason, in a microservices architecture, we recommend using multiple application packages. Put one or more closely related service types into a single application type. For example, put service types in the same application type if your team is responsible for a set of services that have one of these attributes:

- They run for the same duration and need to be updated at the same time.
- They have the same lifecycle.
- They share resources such as dependencies or configuration.

Service Fabric programming models

When you add a microservice to a Service Fabric application, decide whether it has state or data that needs to be made highly available and reliable. If so, can it store data externally or is the data contained as part of the service? Choose a stateless service if you don't need to store data or you want to store data in external storage. Consider choosing a stateful service if one of these statements applies:

- You want to maintain state or data as part of the service. For example, you need that data to reside in memory close to the code.
- You can't tolerate a dependency on an external store.

If you have existing code that you want to run on Service Fabric, you can run it as a *guest executable*: an arbitrary executable that runs as a service. Alternatively, you can package the executable in a container that has all the dependencies that you need for deployment.

Service Fabric models both containers and guest executables as stateless services. For guidance on choosing a model, see [Service Fabric programming model overview](#).

You're responsible for maintaining the environment in which a guest executable runs. For example, suppose that a guest executable requires Python. If the executable is not self-contained, you need to make sure that the required version of Python is pre-installed in the environment. Service Fabric doesn't manage the environment. Azure offers multiple mechanisms to set up the environment, including custom virtual machine images and extensions.

To access a guest executable through a reverse proxy, make sure that you've added the `UriScheme` attribute to the `Endpoint` element in the guest executable's service manifest.

XML

```
<Endpoints>
  <Endpoint Name="MyGuestExeTypeEndpoint" Port="8090" Protocol="http"
UriScheme="http" PathSuffix="api" Type="Input"/>
</Endpoints>
```

If the service has additional routes, specify the routes in the `PathSuffix` value. The value should not be prefixed or suffixed with a slash (/). Another way is to add the route in the service name.

XML

```
<Endpoints>
  <Endpoint Name="MyGuestExeTypeEndpoint" Port="8090" Protocol="http"
PathSuffix="api" Type="Input"/>
</Endpoints>
```

For more information, see:

- [Package an application](#)
- [Package and deploy an existing executable to Service Fabric](#)

API gateway

An [API gateway](#) (ingress) sits between external clients and the microservices. It acts as a reverse proxy, routing requests from clients to microservices. It might also perform cross-cutting tasks such as authentication, SSL termination, and rate limiting.

We recommend Azure API Management for most scenarios, but [Traefik](#) is a popular open-source alternative. Both technology options are integrated with Service Fabric.

- **API Management.** Exposes a public IP address and routes traffic to your services. It runs in a dedicated subnet in the same virtual network as the Service Fabric cluster.

API Management can access services in a node type that's exposed through a load balancer with a private IP address. This option is available only in the Premium and Developer tiers of API Management. For production workloads, use the Premium tier. Pricing information is described in [API Management pricing](#).

For more information, see [Service Fabric with Azure API Management overview](#).

- **Traefik.** Supports features such as routing, tracing, logs, and metrics. Traefik runs as a stateless service in the Service Fabric cluster. Service versioning can be supported through routing.

For information on how to set up Traefik for service ingress and as the reverse proxy within the cluster, see [Azure Service Fabric Provider](#) on the Traefik website. For more information about using Traefik with Service Fabric, see the blog post [Intelligent routing on Service Fabric with Traefik](#).

Traefik, unlike Azure API Management, does not have functionality to resolve the partition of a stateful service (with more than one partition) to which a request is routed. For more information, see [Add a matcher for partitioning services](#).

Other API management options include [Azure Application Gateway](#) and [Azure Front Door](#). You can use these services in conjunction with API Management to perform tasks such as routing, SSL termination, and firewall.

Interservice communication

To facilitate service-to-service communication, consider the following recommendations:

- **Communication protocol.** In a microservices architecture, services need to communicate with each other with minimum coupling at runtime. To enable language-agnostic communication, HTTP is an industry standard with a wide range

of tools and HTTP servers that are available in different languages. Service Fabric supports all of those tools and servers.

For most workloads, we recommend that you use HTTP instead of the service remoting that's built in to Service Fabric.

- **Service discovery.** To communicate with other services within a cluster, a client service needs to resolve the target service's current location. In Service Fabric, services can move between nodes and cause the service endpoints to change dynamically.

To avoid connections to stale endpoints, you can use the naming service in Service Fabric to retrieve updated endpoint information. However, Service Fabric also provides a built-in [reverse proxy service](#) that abstracts the naming service. We recommend this option for service discovery as a baseline for most scenarios, because it's easier to use and results in simpler code.

Other options for interservice communication include:

- [Traefik](#) ↗ for advanced routing.
- [DNS](#) for compatibility scenarios where a service expects to use DNS.
- The [ServicePartitionClient<TCommunicationClient>](#) class, which caches service endpoints. It can enable better performance, because calls go directly between services without intermediaries or custom protocols.

Scalability

Service Fabric supports scaling these cluster entities:

- Scaling the number of nodes for each node type
- Scaling services

This section is focused on autoscaling. You can choose to manually scale in situations where it's appropriate. For example, manual intervention might be required to set the number of instances.

Initial cluster configuration for scalability

When you create a Service Fabric cluster, provision the node types based on your security and scalability needs. Each node type is mapped to a virtual machine scale set and can be scaled independently.

- Create a node type for each group of services that have different scalability or resource requirements. Start by provisioning a node type (which becomes the [primary node type](#)) for the Service Fabric system services. Create separate node types to run your public or front-end services. Create other node types as necessary for your back end and private or isolated services. Specify [placement constraints](#) so that the services are deployed only to the intended node types.
- Specify the *durability tier* for each node type. The durability tier represents the ability of Service Fabric to influence updates and maintenance operations in virtual machine scale sets. For production workloads, choose a Silver or higher durability tier. For information about each tier, see [Durability characteristics of the cluster](#).
- If you're using the Bronze durability tier, certain operations require manual steps. Node types with the Bronze durability tier require additional steps during scale-in. For more information on scaling operations, see [this guide](#).

Scaling nodes

Service Fabric supports autoscaling for scale-in and scale-out. You can configure each node type for autoscaling independently.

Each node type can have a maximum of 100 nodes. Start with a smaller set of nodes, and add more nodes depending on your load. If you require more than 100 nodes in a node type, you'll need to add more node types. For details, see [Service Fabric cluster capacity planning considerations](#). A virtual machine scale set does not scale instantaneously, so consider that factor when you set up autoscale rules.

To support automatic scale-in, configure the node type to have the Silver or Gold durability tier. This configuration makes sure that scaling in is delayed until Service Fabric finishes relocating services. It also makes sure that the virtual machine scale sets inform Service Fabric that the VMs are removed, not just down temporarily.

For more information about scaling at the node or cluster level, see [Scaling Azure Service Fabric clusters](#).

Scaling services

Stateless and stateful services apply different approaches to scaling.

For a stateless service (autoscaling):

- Use the average partition load trigger. This trigger determines when the service is scaled in or out, based on a load threshold value that's specified in the scaling policy. You can also set how often the trigger is checked. See [Average partition](#)

load trigger with instance-based scaling. This approach allows you to scale up to the number of available nodes.

- Set `InstanceCount` to -1 in the service manifest, which tells Service Fabric to run an instance of the service on every node. This approach enables the service to scale dynamically as the cluster scales. As the number of nodes in the cluster changes, Service Fabric automatically creates and deletes service instances to match.

ⓘ Note

In some cases, you might want to manually scale your service. For example, if you have a service that reads from Azure Event Hubs, you might want a dedicated instance to read from each event hub partition. That way, you can avoid concurrent access to the partition.

For a stateful service, scaling is controlled by the number of partitions, the size of each partition, and the number of partitions or replicas running on a machine:

- If you're creating partitioned services, make sure that each node gets adequate replicas for even distribution of the workload without causing resource contentions. If you add more nodes, Service Fabric distributes the workloads onto the new machines by default. For example, if there are 5 nodes and 10 partitions, Service Fabric will place two primary replicas on each node by default. If you scale out the nodes, you can achieve greater performance because the work is evenly distributed across more resources.

For information about scenarios that take advantage of this strategy, see [Scaling in Service Fabric](#).

- Adding or removing partitions is not well supported. Another option that's commonly used to scale is to dynamically create or delete services or whole application instances. An example of that pattern is described in [Scaling by creating or removing new named services](#).

For more information, see:

- [Scale a Service Fabric cluster in or out by using autoscale rules or manually](#)
- [Scale a Service Fabric cluster programmatically](#)
- [Scale out a Service Fabric cluster by adding a virtual machine scale set](#)

Using metrics to balance load

Depending on how you design the partition, you might have nodes with replicas that get more traffic than others. To avoid this situation, partition the service state so that it's distributed across all partitions. Use the range partitioning scheme with a good hash algorithm. See [Get started with partitioning](#).

Service Fabric uses metrics to know how to place and balance services within a cluster. You can specify a default load for each metric associated with a service when that service is created. Service Fabric then takes that load into account when placing the service, or whenever the service needs to move (for example, during upgrades), to balance the nodes in the cluster.

The initially specified default load for a service will not change over the lifetime of the service. To capture changing metrics for a service, we recommend that you monitor your service and then report the load dynamically. This approach allows Service Fabric to adjust the allocation based on the reported load at a given time. Use the [IServicePartition.ReportLoad](#) method to report custom metrics. For more information, see [Dynamic load](#).

Availability

Place your services in a node type other than the primary node type. The Service Fabric system services are always deployed to the primary node type. If your services are deployed to the primary node type, they might compete with (and interfere with) system services for resources. If a node type is expected to host stateful services, make sure that there are at least five node instances and that you select the Silver or Gold durability tier.

Consider constraining the resources of your services. See [Resource governance mechanism](#).

Here are common considerations:

- Don't mix services that are resource governed and services that are not resource governed on the same node type. The non-governed services might consume too many resources and affect the governed services. Specify [placement constraints](#) to make sure that those types of services don't run on the same set of nodes. (This is an example of the [Bulkhead pattern](#).)
- Specify the CPU cores and memory to reserve for a service instance. For information about usage and limitations of resource governance policies, see [Resource governance](#).

To avoid a single point of failure (SPOF), make sure that every service's target instance or replica count is greater than one. The largest number that you can use as a service

instance or replica count equals the number of nodes that constrain the service.

Make sure that every stateful service has at least two active secondary replicas. We recommend five replicas for production workloads.

For more information, see [Availability of Service Fabric services](#).

Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Here are some key points for securing your application on Service Fabric.

Virtual network

Consider defining subnet boundaries for each virtual machine scale set to control the flow of communication. Each node type has its own virtual machine scale set in a subnet within the Service Fabric cluster's virtual network. You can add network security groups (NSGs) to the subnets to allow or reject network traffic. For example, with front-end and back-end node types, you can add an NSG to the back-end subnet to accept inbound traffic only from the front-end subnet.

When you're calling external Azure services from the cluster, use [virtual network service endpoints](#) if the Azure service supports it. Using a service endpoint secures the service to only the cluster's virtual network.

For example, if you're using Azure Cosmos DB to store data, configure the Azure Cosmos DB account with a service endpoint to allow access only from a specific subnet. See [Access Azure Cosmos DB resources from virtual networks](#).

Endpoints and interservice communication

Don't create an unsecured Service Fabric cluster. If the cluster exposes management endpoints to the public internet, anonymous users can connect to it. Unsecured clusters are not supported for production workloads. See [Service Fabric cluster security scenarios](#).

To help secure your interservice communications:

- Consider enabling HTTPS endpoints in your ASP.NET Core or Java web services.
- Establish a secure connection between the reverse proxy and services. For details, see [Connect to a secure service](#).

If you're using an [API gateway](#), you can [offload authentication](#) to the gateway. Make sure that the individual services can't be reached directly (without the API gateway) unless additional security is in place to authenticate messages.

Don't expose the Service Fabric reverse proxy publicly. Doing so causes all services that expose HTTP endpoints to be addressable from outside the cluster. That will introduce security vulnerabilities and potentially expose additional information outside the cluster unnecessarily. If you want to access a service publicly, use an API gateway. The [API gateway](#) section later in this article mentions some options.

Remote Desktop is useful for diagnostic and troubleshooting, but be sure to close it. Leaving it open causes a security hole.

Secrets and certificates

Store secrets, such as connection strings to data stores, in a key vault. The key vault must be in the same region as the virtual machine scale set. To use a key vault:

1. Authenticate the service's access to the key vault.

Enable [managed identity](#) on the virtual machine scale set that hosts the service.

2. Store your secrets in the key vault.

Add secrets in a format that can be translated to a key/value pair. For example, use `CosmosDB--AuthKey`. When the configuration is built, the double hyphen (`--`) is converted into a colon (`:`).

3. Access those secrets in your service.

Add the key vault URI in your `appSettings.json` file. In your service, add the configuration provider that reads from the key vault, builds the configuration, and accesses the secret from the built configuration.

Here's an example where the Workflow service stores a secret in the key vault in the format `CosmosDB--Database`.

C#

```
namespace Fabrikam.Workflow.Service
{
    public class ServiceStartup
    {
        public static void ConfigureServices(StatelessServiceContext
context, IServiceCollection services)
        {
```

```

var preConfig = new ConfigurationBuilder()
    ...
    .AddJsonFile(context, "appsettings.json");

var config = preConfig.Build();

if (config["AzureKeyVault:KeyVaultUri"] is var keyVaultUri &&
!string.IsNullOrWhiteSpace(keyVaultUri))
{
    preConfig.AddAzureKeyVault(keyVaultUri);
    config = preConfig.Build();
}
}
}

```

To access the secret, specify the secret name in the built configuration.

C#

```

if(builtConfig["CosmosDB:Database"] is var database &&
!string.IsNullOrEmpty(database))
{
    // Use the secret.
}

```

Don't use client certificates to access Service Fabric Explorer. Instead, use Microsoft Entra ID. See [Azure services that support Microsoft Entra authentication](#).

Don't use self-signed certificates for production.

Protection of data at rest

If you attached data disks to the virtual machine scale sets of the Service Fabric cluster and your services save data on those disks, you must encrypt the disks. For more information, see [Encrypt OS and attached data disks in a virtual machine scale set with Azure PowerShell \(preview\)](#).

For more information about securing Service Fabric, see:

- [Azure Service Fabric security overview](#)
- [Azure Service Fabric security best practices](#)
- [Azure Service Fabric security checklist](#)

Resiliency

To recover from failures and maintain a fully functioning state, the application must implement certain resiliency patterns. Here are some common patterns:

- [Retry](#): To handle errors that you expect to be transient, such as resources that are temporarily unavailable.
- [Circuit breaker](#): To address faults that might take longer to fix.
- [Bulkhead](#): To isolate resources for each service.

This reference implementation uses [Polly](#), an open-source option, to implement all of those patterns.

Monitoring

Before you explore the monitoring options, we recommend that you read [this article about diagnosing common scenarios with Service Fabric](#). You can think of monitoring data in these sets:

- Application metrics and logs
- Service Fabric health and event data
- Infrastructure metrics and logs
- Metrics and logs for dependent services

These are the two main options for analyzing that data:

- Application Insights
- Log Analytics

You can use Azure Monitor to set up dashboards for monitoring and to send alerts to operators. Some third-party monitoring tools are also integrated with Service Fabric, such as Dynatrace. For details, see [Azure Service Fabric monitoring partners](#).

Application metrics and logs

Application telemetry provides data that can help you monitor the health of your service and identify problems. To add traces and events in your service:

- Use [Microsoft.Extensions.Logging](#) if you're developing your service with ASP.NET Core. For other frameworks, use a logging library of your choice, such as Serilog.
- Add your own instrumentation by using the [TelemetryClient](#) class in the SDK, and view the data in Application Insights. See [Add custom instrumentation to your application](#).
- Log Event Tracing for Windows (ETW) events by using [EventSource](#). This option is available by default in a Visual Studio Service Fabric solution.

Application Insights provides a lot of built-in telemetry: requests, traces, events, exceptions, metrics, dependencies. If your service exposes HTTP endpoints, enable Application Insights by calling the `UseApplicationInsights` extension method for `Microsoft.AspNetCore.Hosting.IWebHostBuilder`. For information about instrumenting your service for Application Insights, see these articles:

- [Tutorial: Monitor and diagnose an ASP.NET Core application on Service Fabric using Application Insights](#)
- [Application Insights for ASP.NET Core](#)
- [Application Insights .NET SDK](#)
- [Application Insights SDK for Service Fabric ↗](#)

To view the traces and event logs, use [Application Insights](#) as one of the sinks for structured logging. Configure Application Insights with your instrumentation key by calling the `AddApplicationInsights` extension method. In this example, the instrumentation key is stored as a secret in the key vault.

C#

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddApplicationInsights(hostingContext.Configuration
["ApplicationInsights:InstrumentationKey"]);
})
```

If your service doesn't expose HTTP endpoints, you need to write a custom extension that sends traces to Application Insights. For an example, see the Workflow service in the [reference implementation ↗](#).

ASP.NET Core services use the [ILogger interface](#) for application logging. To make these application logs available in Azure Monitor, send the `ILogger` events to Application Insights. Application Insights can add correlation properties to `ILogger` events, which is useful for visualizing distributed tracing.

For more information, see:

- [Application logging](#)
- [Add logging to your Service Fabric application](#)

Service Fabric health and event data

Service Fabric telemetry includes health metrics and events about the operation and performance of a Service Fabric cluster and its entities: its nodes, applications, services,

partitions, and replicas. Health and event data can come from:

- [EventStore](#). This stateful system service collects events related to the cluster and its entities. Service Fabric uses EventStore to write [Service Fabric events](#) to provide information about your cluster for status updates, troubleshooting, and monitoring. EventStore can also correlate events from different entities at a given time to identify problems in the cluster. The service exposes those events through a REST API.

For information about how to query the EventStore APIs, see [Query EventStore APIs for cluster events](#). You can view the events from EventStore in Log Analytics by configuring your cluster with the Azure Diagnostics extension for Windows (WAD).

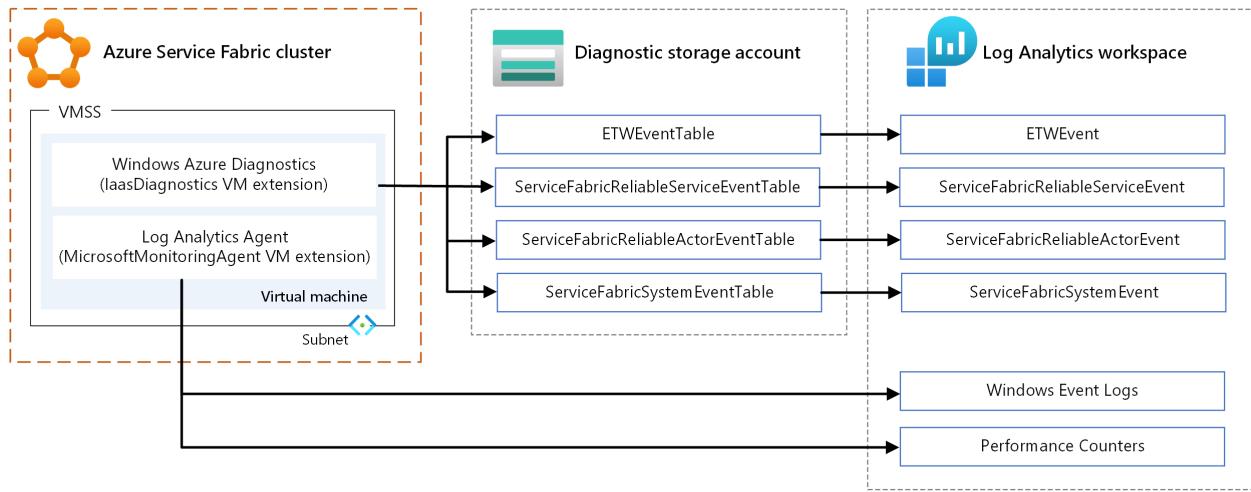
- [HealthStore](#). This stateful service provides a snapshot of the current health of the cluster. It aggregates all health data reported by entities in a hierarchy. The data is visualized in [Service Fabric Explorer](#). HealthStore also monitors application upgrades. You can use health queries in PowerShell, a .NET application, or REST APIs. See [Introduction to Service Fabric health monitoring](#).
- [Custom health reports](#). Consider implementing internal watchdog services that can periodically report custom health data, such as faulty states of running services. You can read the health reports in Service Fabric Explorer.

Infrastructure metrics and logs

Infrastructure metrics help you understand resource allocation in the cluster. Here are the main options for collecting this information:

- [WAD](#). Collect logs and metrics at the node level on Windows. You can use WAD by configuring the IaaSDiagnostics VM extension on any virtual machine scale set that's mapped to a node type to collect diagnostic events. These events can include Windows event logs, performance counters, ETW/manifest system and operational events, and custom logs.
- [Log Analytics agent](#). Configure the MicrosoftMonitoringAgent VM extension to send Windows event logs, performance counters, and custom logs to Log Analytics.

There's some overlap in the types of metrics collected through the preceding mechanisms, such as performance counters. Where there's overlap, we recommend using the Log Analytics agent. Because the Log Analytics agent doesn't use Azure storage, latency is low. Also, the performance counters in IaaSDiagnostics can't be fed into Log Analytics easily.



For information about using VM extensions, see [Azure virtual machine extensions and features](#).

To view the data, configure Log Analytics to display the data collected through WAD. For information about how to configure Log Analytics to read events from a storage account, see [Set up Log Analytics for a cluster](#).

You can also view performance logs and telemetry data related to a Service Fabric cluster, workloads, network traffic, pending updates, and more. See [Performance monitoring with Log Analytics](#).

The [Service Map solution in Log Analytics](#) provides information about the topology of the cluster (that is, the processes running in each node). Send the data in the storage account to [Application Insights](#). There might be some delay in getting data into Application Insights. If you want to see the data in real time, consider configuring [Event Hubs](#) by using sinks and channels. For more information, see [Event aggregation and collection using WAD](#).

Dependent service metrics

- [Application Map in Application Insights](#) provides the topology of the application by using HTTP dependency calls made between services, with the installed Application Insights SDK.
- [Service Map in Log Analytics](#) provides information about inbound and outbound traffic from and to external services. Service Map integrates with other solutions, such as updates or security.
- Custom watchdogs can report error conditions on external services. For example, the service can provide an error health report if it can't access an external service or data storage (Azure Cosmos DB).

Distributed tracing

In a microservices architecture, several services often participate to complete a task. The telemetry from each of those services is correlated through context fields (like operation ID and request ID) in a distributed trace.

By using [Application Map](#) in Application Insights, you can build the view of distributed logical operations and visualize the entire service graph of your application. You can also use transaction diagnostics in Application Insights to correlate server-side telemetry. For more information, see [Unified cross-component transaction diagnostics](#).

It's also important to correlate tasks that are dispatched asynchronously by using a queue. For details about sending correlation telemetry in a queue message, see [Queue instrumentation](#).

For more information, see:

- [Performing a query across multiple resources](#)
- [Telemetry correlation in Application Insights](#)

Alerts and dashboards

Application Insights and Log Analytics support an [extensive query language](#) (Kusto query language) that lets you retrieve and analyze log data. Use the queries to create data sets and visualize them in diagnostics dashboards.

Use Azure Monitor alerts to notify system admins when certain conditions occur in specific resources. The notification could be an email, an Azure function, or a webhook, for example. For more information, see [Alerts in Azure Monitor](#).

[Log search alert rules](#) allow you to define and run a Kusto query against a Log Analytics workspace at regular intervals. An alert is created if the query result matches a certain condition.

Cost optimization

Use the [Azure pricing calculator](#) to estimate costs. Other considerations are described in the [cost optimization pillar of the Microsoft Azure Well-Architected Framework](#).

Here are some points to consider for some of the services used in this architecture.

Azure Service Fabric

You're charged for the compute instances, storage, networking resources, and IP addresses that you choose when creating a Service Fabric cluster. There are deployment charges for Service Fabric.

Virtual machine scale sets

In this architecture, microservices are deployed into nodes that are virtual machine scale sets. You're charged for the Azure VMs that are deployed as part of the cluster and underlying infrastructure resources, such as storage and networking. There are no incremental charges for the virtual machine scale sets themselves.

Azure API Management

Azure API Management is a gateway to route the requests from clients to your services in the cluster.

There are various pricing options. The Consumption option is charged on a pay-per-use basis and includes a gateway component. Based on your workload, choose an option described in [API Management pricing](#).

Application Insights

You can use Application Insights to collect telemetry for all services and to view the traces and event logs in a structured way. The pricing for Application Insights is a pay-as-you-go model that's based on ingested data volume and options for data retention. For more information, see [Manage usage and cost for Application Insights](#).

Azure Monitor

For Azure Monitor Log Analytics, you're charged for data ingestion and retention. For more information, see [Azure Monitor pricing](#).

Azure Key Vault

You use Azure Key Vault to store the instrumentation key for Application Insights as a secret. Azure offers Key Vault in two service tiers. If you don't need HSM-protected keys, choose the Standard tier. For information about the features in each tier, see [Key Vault pricing](#).

Azure DevOps Services

This reference architecture uses Azure Pipelines for deployment. The Azure Pipelines service allows a free Microsoft-hosted job with 1,800 minutes per month for CI/CD and one self-hosted job with unlimited minutes per month. Extra jobs have charges. For more information, see [Azure DevOps Services pricing](#).

For DevOps considerations in a microservices architecture, see [CI/CD for microservices](#).

To learn how to deploy a container application with CI/CD to a Service Fabric cluster, see [this tutorial](#).

Deploy this scenario

To deploy the reference implementation for this architecture, follow the steps in the [GitHub repo](#).

Next steps

- [Training: Introduction to Azure Service Fabric](#)
- [Overview of Azure Service Fabric](#)
- [API Management documentation](#)
- [What is Azure Pipelines?](#)

Related resources

- [Using domain analysis to model microservices](#)
- [Designing a microservices architecture](#)

Deploy microservices with Azure Container Apps

Azure Container Apps Azure Cosmos DB Azure Service Bus

This example scenario shows an example of an existing workload that was originally designed to run on Kubernetes can instead run in Azure Container Apps. Azure Container Apps is well-suited for brownfield workloads where teams are looking to simplify complex infrastructure and container orchestration. The example workload runs a containerized microservices application.

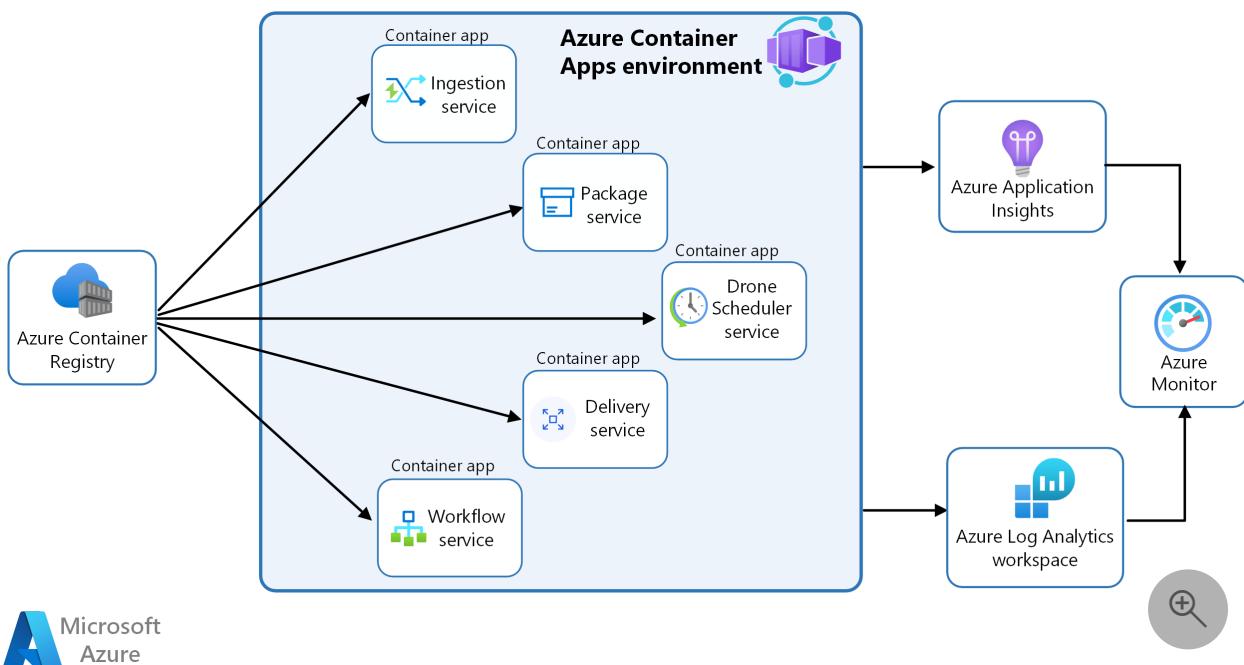
The example takes the workload used in [Microservices architecture on Azure Kubernetes Service](#) and rehosts it in Azure Container Apps as its application platform.

Tip



The architecture is backed by an [example implementation](#) that illustrates some of design choices described in this article.

Architecture



Download a [Visio file](#) of this architecture.

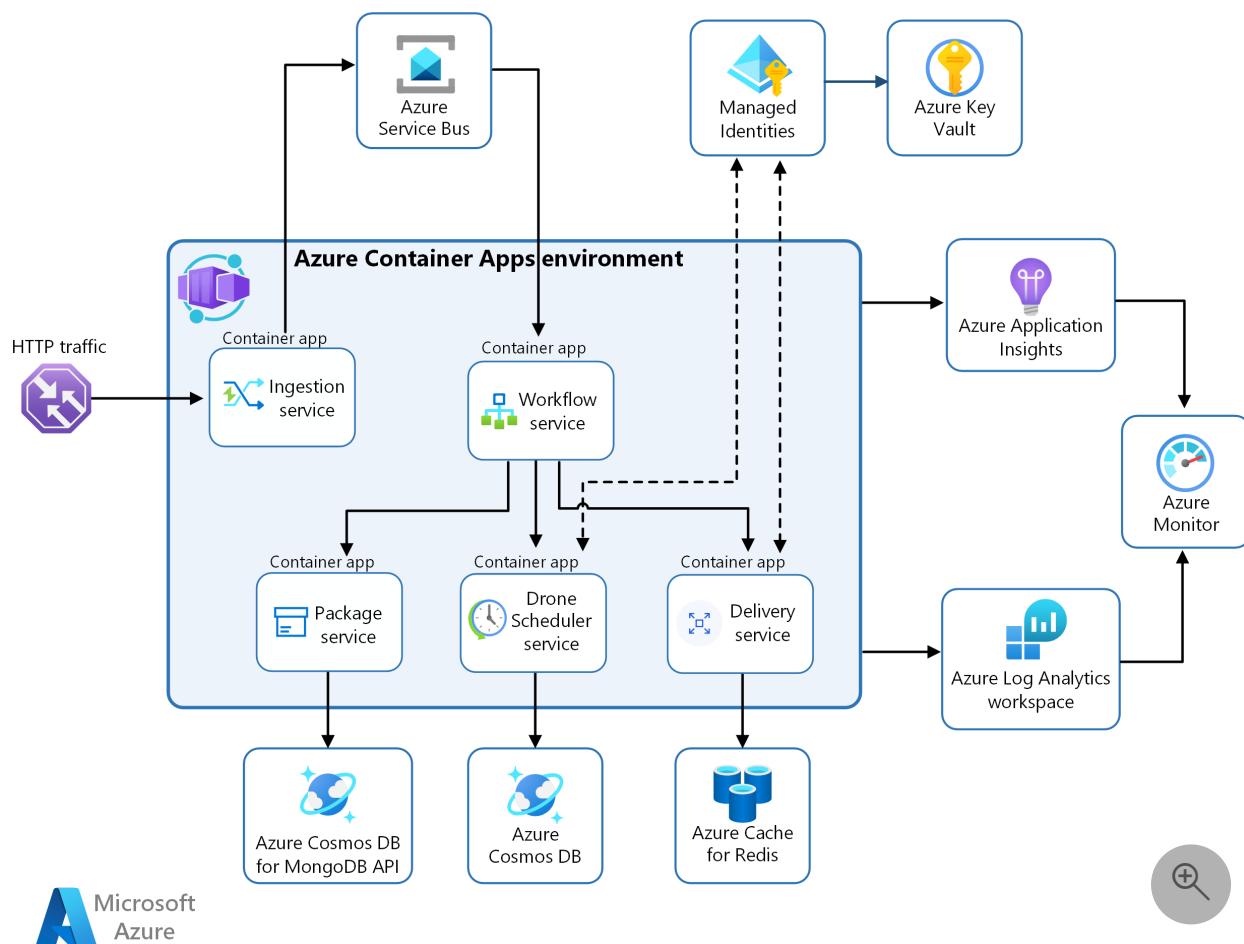
In this scenario, the container images are sourced from Azure Container Registry and deployed to a Container Apps environment.

The services sharing the same environment benefit from:

- Internal ingress and service discovery
- A single Log Analytics workspace for runtime logging
- Secure management of secrets and certificates

The workflow service container app is running in single revision mode. A container app running in single revision mode will have a single revision that is backed by zero-many replicas. A replica is composed of the application container and any required sidecar containers. This example isn't making use of sidecar containers, therefore each container app replica represents a single container. Since this example doesn't employ scaling, there will be only one replica running for each container app.

The workflow uses a hybrid approach to managing secrets. Managed identities are used in the services where such implementation required no code changes. The Drone Scheduler and Delivery services use user-assigned managed identities to authenticate with Azure Key Vault to access the secrets stored there. The remaining services store secrets via Container Apps service at the application level.



This diagram illustrates the runtime architecture for the solution.

Download a [Visio file](#) of this architecture.

Dataflow

1. **Ingestion service:** Receives client requests, buffers them and sends them via Azure Service Bus to the workflow service.
2. **Workflow service:** Consumes messages from Azure Service Bus and dispatches them to underlying services.
3. **Package service:** Manages packages.
4. **Drone scheduler service:** Schedules drones and monitors drones in flight.
5. **Delivery service:** Manages deliveries that are scheduled or in-transit.

Components

The drone delivery service uses a series of Azure services in concert with one another.

Azure Container Apps

[Azure Container Apps](#) is the primary component.

Many of the complexities of the previous AKS architecture are replaced by these features:

- Built-in service discovery
- Fully managed HTTP and HTTP/2 endpoints
- Integrated load balancing
- Logging and monitoring
- Autoscaling based on HTTP traffic or events powered by KEDA
- Application upgrades and versioning

External storage and other components

[Azure Key Vault](#) service for securely storing and accessing secrets, such as API keys, passwords, and certificates.

[Azure Container Registry](#) stores private container images. You can also use other container registries like Docker Hub.

[Azure Cosmos DB](#) stores data using the open-source [Azure Cosmos DB for MongoDB](#). Microservices are typically stateless and write their state to external data stores. Azure Cosmos DB is a NoSQL database with open-source APIs for MongoDB and Cassandra.

[Azure Service Bus](#) offers reliable cloud messaging as a service and simple hybrid integration. Service Bus supports asynchronous messaging patterns that are common with microservices applications.

[Azure Cache for Redis](#) adds a caching layer to the application architecture to improve speed and performance for heavy traffic loads.

[Azure Monitor](#) collects and stores metrics and logs from the application. Use this data to monitor the application, set up alerts and dashboards, and do root cause analysis of failures. This scenario uses a Log Analytics workspace for comprehensive monitoring of the infrastructure and application.

[Application Insights](#) provides extensible application performance management (APM) and monitoring for the services. Each service is instrumented with the Application Insights SDK to monitor the app and direct the data to Azure Monitor.

[Bicep Templates](#) to configure and deploy the applications.

Alternatives

An alternative scenario of this example is the Fabrikam Drone Delivery application using Kubernetes, which is available on GitHub in the [Azure Kubernetes Service \(AKS\) Fabrikam Drone Delivery](#) repository.

Scenario details

Your business can simplify the deployment and management of microservice containers by using Azure Container Apps. Container Apps provides a fully managed serverless environment for building and deploying modern applications.

Fabrikam, Inc. (a fictional company) has implemented a drone delivery application where users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time.

The microservices application was deployed to an Azure Kubernetes Service (AKS) cluster. But, the Fabrikam team wasn't taking advantage of the advanced or platform-specific AKS features. They eventually migrated the application to Azure Container Apps without much overhead. By porting their solution to Azure Container Apps, Fabrikam was able to:

- Migrate the application nearly as-is: Very minimal code changes were required when moving their application from AKS to Azure Container Apps.

- Deploy both infrastructure and the workload with Bicep templates: No Kubernetes YAML manifests were needed to deploy their application containers.
- Expose the application through managed ingress: Built-in support for external, https-based ingress to expose the Ingestion Service removed the need for configuring their own ingress.
- Pull container images from ACR: Azure Container Apps doesn't require a specific base image or registry.
- Manage application lifecycle: The revision feature supports running multiple revisions of a particular container app and traffic-splitting across them for A/B testing or Blue/Green deployment scenarios.
- Use managed identity: The Fabrikam team was able to use a managed identity to authenticate with Azure Key Vault and Azure Container Registry.

Potential use cases

- Deploy a brownfield microservice-based application into a platform as a service (PaaS) offering to avoid the operational complexity of managing a container orchestrator.
- Optimize operations and management by migrating containerized services to a platform that supports native scale-to-zero.
- Execute a long-running background process, such as the workflow service in single revision mode.

Other common uses of Container Apps include:

- Running containerized workloads on a serverless, consumption-based platform.
- Autoscaling applications based on HTTP/HTTPS traffic and/or Event-driven triggers supported by KEDA
- Minimizing maintenance overhead for containerized applications
- Deploying API endpoints
- Hosting background processing applications
- Handling event-driven processing

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Availability

Container Apps allows you to more easily deploy, manage, maintain and monitor the applications. Availability can be ensured by these key features:

- Revisions help you deploy application updates with zero downtime. You can use revisions to manage the deployment of application updates and split traffic between the revisions to support blue/green deployments and A/B testing (not currently used in this example workload).
- With Container Apps end-to-end observability features, you have a holistic view of your running applications. Container Apps is integrated with Azure Monitor and Log Analytics, which allows you to track container app execution, and set alerts based on metrics and events.
- When an app unexpectedly terminates, the Container Apps service automatically restarts it.
- You can enable autoscaling rules to meet demand as traffic and workloads increase.
- Performance is optimized by the dynamic load balancing features of Container Apps (not currently used in this example workload).

Operational excellence

Operational excellence covers the operations processes that deploy an application and keep it running in production. For more information, see [Overview of the operational excellence pillar](#).

To achieve operational excellence, the Container Apps service offers these features:

- GitHub Actions integration for setting up automated CI/CD deployments.
- Multi-revision mode with traffic splitting for testing changes to your application code and scale rules.
- Integration with Azure Monitor and Log Analytics to provide insight into your containerized application.

Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

Performance considerations in this solution:

- The workload is distributed among multiple microservice applications.

- Each microservice is independent, sharing nothing with the other microservices, so that they can independently scale.
- Autoscaling can be enabled as the workload increases.
- Requests are dynamically load balanced.
- Metrics, including CPU and memory utilization, bandwidth information and storage utilization, are available through Azure Monitor.
- Log analytics provides log aggregation to gather information across each Container Apps environment.

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

Container Apps will attempt to restart failing containers and abstracts away hardware from users. Transient failures and high-availability of backing compute resources are handled by Microsoft.

Performance monitoring through Log Analytics and Azure Monitor allows you to evaluate the application under load. Metrics and logging information give you the data needed to recognize trends to prevent failures and perform root-cause analysis of failures when they occur.

Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Secrets

- Your container app can store and retrieve sensitive values as secrets. After a secret is defined for the container app, it's available for use by the application and any associated scale rules. If you're running in multi-revision mode, all revisions will share the same secrets. Because secrets are considered an application-scope change, if you change the value of a secret, a new revision isn't created. However, for any running revisions to load the new secret value, you'll need to restart them. In this scenario, application and environment variable values are used.
- Environment variables: sensitive values can be securely stored at the application level. When environment variables are changed, the container app will spawn a new revision.

Network security

- Ingress: To limit external access, only the Ingestion service is configured for external ingress. The backend services are accessible only through the internal virtual network in the Container Apps environment. Only expose services to the Internet where required. Because this architecture uses the built-in external ingress feature, this solution does not offer the ability to completely position your ingress point behind a web application firewall (WAF) or to include it in DDoS Protection plans. All web facing workloads should be fronted with a web application firewall.
- Virtual network: When you create an environment, you can provide a custom virtual network; otherwise, a virtual network is automatically generated and managed by Microsoft. You cannot manipulate this Microsoft-managed virtual network, such as by adding network security groups (NSGs) or force tunneling traffic to a egress firewall. This example uses an automatically generated virtual network.

For more network topology options, see [Networking architecture in Azure Container Apps](#).

Workload identities

- Container Apps supports Microsoft Entra managed identities allowing your app to authenticate itself to other resources protected by Microsoft Entra ID, such as Azure Key Vault, without managing credentials in your container app. A container app can use system-assigned, user-assigned, or both types of managed identities. For services that don't support AD authentication, you should store secrets in Azure Key Vault and use a managed identity to access the secrets.
- Use managed identities for Azure Container Registry access. Azure Container Apps supports using a different managed identity for the workload than container registry access, which is recommended when looking to achieve granular access controls on your managed identities.

Cost optimization

- The [Cost section in the Microsoft Azure Well-Architected Framework](#) describes cost considerations. Use the [Azure pricing calculator](#) to estimate costs for your specific scenario.
- Azure Container Apps has consumption based pricing model.
- Azure Container Apps supports scale to zero. When a container app is scaled to zero, there's no charge.

- In this scenario, Azure Cosmos DB and Azure Cache for Redis are the main cost drivers.

Deploy this scenario

Follow the steps in the README.md in the [Azure Container Apps example scenario](#) repository to deploy this scenario.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Catherine Bundy](#) | Technical Writer

Next steps

- [Azure Container Apps Documentation](#)
- [Azure Kubernetes Service \(AKS\) Fabrikam Drone Delivery GitHub repo](#)

Related resources

- [Build microservices on Azure](#)
- [Design a microservices architecture](#)
- [CI/CD for AKS apps with Azure Pipelines](#)
- [Advanced Azure Kubernetes Service \(AKS\) microservices architecture](#)
- [Microservices architecture on Azure Kubernetes Service](#)

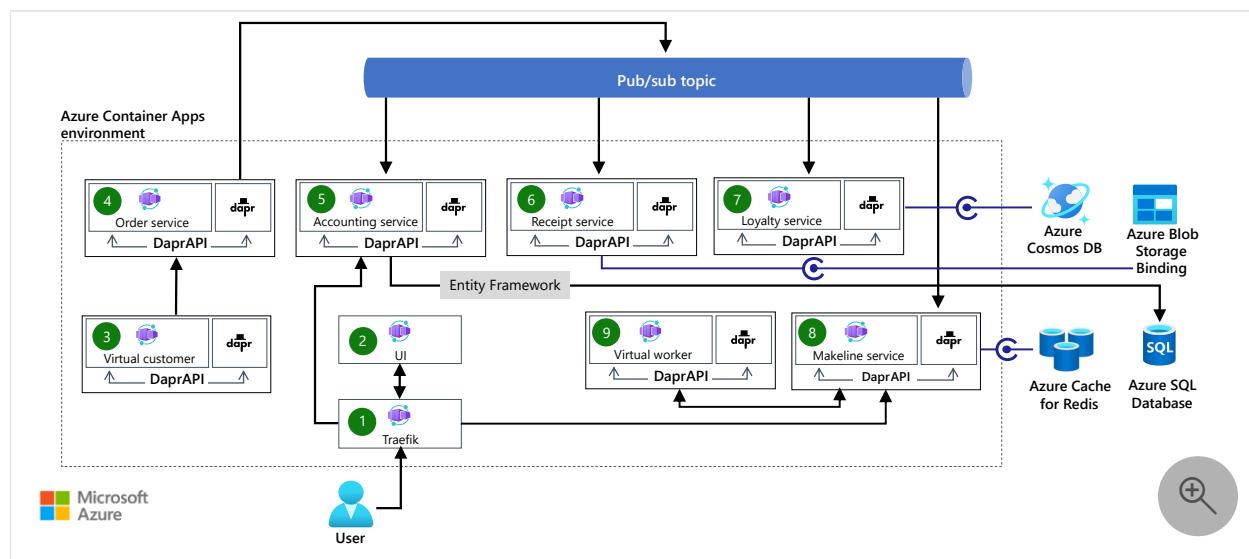
Deploy microservices with Azure Container Apps and Dapr

Azure Container Apps .NET Azure SQL Database Azure Cosmos DB Azure Cache for Redis

This article describes a solution for running an order management system with 10 microservices on Azure Container Apps. The solution also uses microservices best practices through Dapr and event-driven scaling with KEDA.

Dapr and Traefik are trademarks of their respective companies. No endorsement is implied by the use of these marks.

Architecture



Download a [PowerPoint file](#) of this architecture.

Dataflow

This solution uses Bicep templates to execute the deployment of the Reddog order management system and its supporting Azure infrastructure. The architecture is composed of a single Azure Container Apps environment that hosts 10 .NET Core microservice applications. You'll use the .NET Core Dapr SDK to integrate with Azure resources through publish-subscribe (pub/sub) and State and Binding building blocks. Although Dapr typically provides flexibility when you implement components, this solution is based on an opinion. The services also make use of KEDA scale rules to allow for scaling based on event triggers and scale to zero scenarios.

The following list describes each microservice and the Azure Container Apps configuration it deploys with. See the [reddog-code repo on GitHub](#) to view the code.

1. **Traefik**: The basic proxy for routing user requests from the UI to the accounting and Makeline services for the interactive dashboard.
2. **UI**: A dashboard that shows real-time order and aggregated sales data for the Reddog order management system.
3. **Virtual customer**: A customer simulation program that simulates customers placing orders via the order service.
4. **Order service**: A CRUD API to place and manage orders.
5. **Accounting service**: A service that processes, stores, and aggregates order data. It transforms customer orders into meaningful sales metrics that are showcased by the UI.
6. **Receipt service**: An archival program that generates and stores order receipts for auditing and historical purposes.
7. **Loyalty service**: A service that manages the loyalty program by tracking customer reward points based on order spend.
8. **Makeline service**: A service that's responsible for managing a queue of current orders awaiting fulfillment. It tracks the processing and completion of the orders by the virtual worker service.
9. **Virtual worker**: A *worker simulation* program that simulates the completion of customer orders.
10. **Bootstrapper (not shown)**: A service that uses Entity Framework Core to initialize the tables within Azure SQL Database for use with the accounting service.

⋮ Expand table

Service	Ingress	Dapr components	KEDA scale rules
Traefik	External	Dapr not enabled	HTTP
UI	Internal	Dapr not enabled	HTTP

Service	Ingress	Dapr components	KEDA scale rules
Virtual customer	None	Service to service invocation	N/A
Order service	Internal	Pub/sub: Azure Service Bus	HTTP
Accounting service	Internal	Pub/sub: Azure Service Bus	Azure Service Bus topic length, HTTP
Receipt service	Internal	Pub/sub: Azure Service Bus Binding: Azure Blob	Azure Service Bus topic length
Loyalty service	Internal	Pub/sub: Azure Service Bus State: Azure Cosmos DB	Azure Service Bus topic length
Makeline service	Internal	Pub/sub: Azure Service Bus State: Azure Redis	Azure Service Bus topic length, HTTP
Virtual worker	None	Service to service invocation Binding: Cron	N/A

ⓘ Note

You can also execute Bootstrapper in a container app. However, this service is run once to perform the database creation, and then scaled to zero after creating the necessary objects in Azure SQL Database.

Components

This solution uses the following components:

- [Azure resource groups](#) are logical containers for Azure resources. You use a single resource group to structure everything related to this solution in the Azure portal.

- [Azure Container Apps](#) is a fully managed, serverless container service used to build and deploy modern apps at scale. In this solution, you're hosting all 10 microservices on Azure Container Apps and deploying them into a single Container App environment. This environment acts as a secure boundary around the system.
- [Azure Service Bus](#) is a fully managed enterprise message broker complete with queues and publish-subscribe topics. In this solution, use it for the Dapr pub/sub component implementation. Multiple services use this component. The order service publishes messages on the bus, and the Makeline, accounting, loyalty, and receipt services subscribe to these messages.
- [Azure Cosmos DB](#) is a NoSQL, multi-model managed database service. Use it as a Dapr state store component for the loyalty service to store customer's loyalty data.
- [Azure Cache for Redis](#) is a distributed, in-memory, scalable managed Redis cache. It's used as a Dapr state store component for the Makeline Service to store data on the orders that are being processed.
- [Azure SQL Database](#) is an intelligent, scalable, relational database service built for the cloud. Create it for the accounting service, which uses [Entity Framework Core](#) to interface with the database. The Bootstrapper service is responsible for setting up the SQL tables in the database, and then runs once before establishing the connection to the accounting service.
- [Azure Blob Storage](#) stores massive amounts of unstructured data like text or binary files. The receipt service uses Blob Storage via a Dapr output binding to store the order receipts.
- [Traefik](#) is a leading modern reverse proxy and load balancer that makes it easy to deploy microservices. In this solution, use Traefik's dynamic configuration feature to do path-based routing from the UI, which is a Vue.js single-page application (SPA). This configuration also enables direct API calls to the backend services for testing.
- [Azure Monitor](#) enables you to collect, analyze, and act on customer content data from your Azure infrastructure environments. You'll use it with [Application Insights](#) to view the container logs and collect metrics from the microservices.

Alternatives

In this architecture, you deploy a Traefik proxy to enable path-based routing for the Vue.js API. There are many alternative open-source proxies that you can use for this purpose. Two other popular projects are [NGINX](#) and [HAProxy](#).

All Azure infrastructure, except Azure SQL Database, use Dapr components for interoperability. One benefit of Dapr is that you can swap all these components by

changing the container apps deployment configuration. In this case, Azure Service Bus, Azure Cosmos DB, Cache for Redis, and Blob Storage were chosen to showcase some of the 70+ Dapr components available. A list of alternative [pub/sub brokers](#), [state stores](#) and [output bindings](#) are in the Dapr docs.

Scenario details

Microservices are an increasingly popular architecture style that can have many benefits, including high scalability, shorter development cycles, and increased simplicity. You can use containers as a mechanism to deploy microservices applications, and then use a container orchestrator like Kubernetes to simplify operations. There are many factors to consider for large scale microservices architectures. Typically, the infrastructure platform requires significant understanding of complex technologies like the container orchestrators.

[Azure Container Apps](#) is a fully managed serverless container service for running modern applications at scale. It enables you to deploy containerized apps through abstraction of the underlying platform. This way, you won't need to manage a complicated infrastructure. Azure Container Apps is powered by open-source technologies.

This architecture uses Azure Container Apps integration with a managed version of the [Distributed Application Runtime \(Dapr\)](#). Dapr is an open source project that helps developers with the inherent challenges in distributed applications, like state management and service invocation.

Azure Container Apps also provides a managed version of [Kubernetes Event-driven Autoscaling \(KEDA\)](#). KEDA lets your containers autoscale based on incoming events from external services like Azure Service Bus and Azure Cache for Redis.

You can also enable HTTPS ingress in Azure Container Apps without creating more Azure networking resources. You can use [Envoy proxy](#), which also allows traffic splitting scenarios.

To explore how Azure Container Apps compares to other container hosting platforms in Azure, see [Comparing Container Apps with other Azure container options](#).

This article describes a solution for running an order management system with 10 microservices on Azure Container Apps. The solution also uses microservices best practices through Dapr and event-driven scaling with KEDA.

Potential use cases

This solution applies to any organization that uses stateless and stateful microservices for distributed systems. The solution is best for consumer packaged goods and manufacturing industries that have an ordering and fulfillment system.

These other solutions have similar designs:

- Microservices architecture on Azure Kubernetes Service (AKS)
- Microservices architecture on Azure Functions
- Event-driven architectures

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that you can use to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see [Overview of the reliability pillar](#).

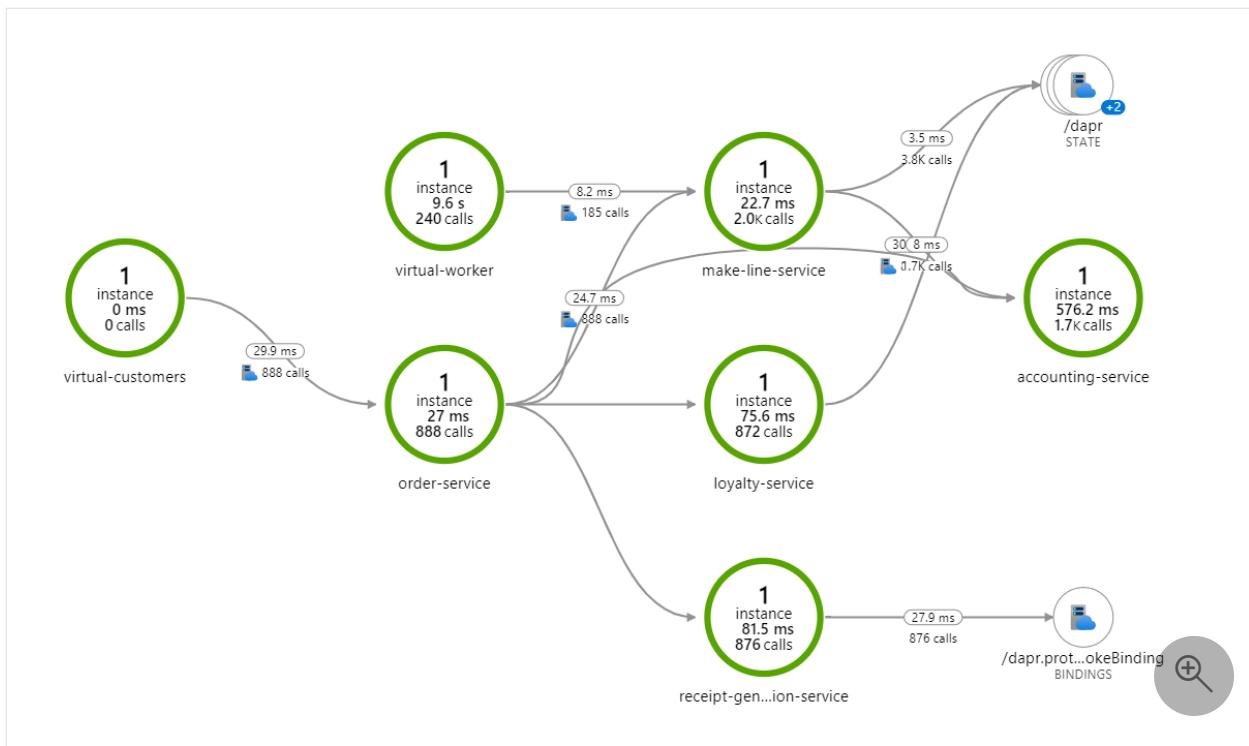
Azure Container Apps runs on Kubernetes behind the scenes. Resiliency mechanisms are built into Kubernetes that monitor and restart containers, or pods, if there are issues. The resiliency mechanisms combine with the built-in load balancer to run multiple replicas of each container app. With this redundancy, the solution can tolerate an instance being unavailable.

You can use Azure Monitor and Application Insights to monitor Azure Container Apps. You can view container logs by navigating in the portal to the **Logs** pane in each container app, and then run the following Kusto query. This example shows logs for the Makeline service app.

Kusto

```
ContainerAppConsoleLogs_CL |  
    where ContainerAppName_s contains "make-line-service" |  
    project TimeGenerated, _timestamp_d, ContainerGroupName_s, Log_s |  
    order by _timestamp_d asc
```

The application map in Application Insights also shows how the services communicate in real time. You can then use them for debugging scenarios. Navigate to the application map under the Application Insights resource to view something like the following.



For more information on monitoring Azure Container Apps, see [Monitor an app in Azure Container Apps](#).

Cost optimization

Optimize costs by looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Use the [Azure pricing calculator](#) to estimate the cost of the services in this architecture.

Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands you place on it in an efficient manner. For more information, see [Performance efficiency pillar overview](#).

This solution relies heavily on the KEDA implementation in Azure Container Apps for event-driven scaling. When you deploy the virtual customer service, it will continuously place orders, which cause the order service to scale up via the HTTP KEDA scaler. As the order service publishes the orders on the service bus, the service bus KEDA scalers cause the accounting, receipt, Makeline, and loyalty services to scale up. The UI and Traefik container apps also configure HTTP KEDA scalers so that the apps scale as more users access the dashboard.

When the virtual customer isn't running, all microservices in this solution scale to zero except for virtual worker and Makeline services. Virtual worker doesn't scale down since it's constantly checking for order fulfillment. For more information on scaling in container apps, see [Set scaling rules in Azure Container Apps](#). For more information on KEDA Scalers, read the [KEDA documentation on Scalers](#).

Deploy this scenario

For deployment instructions, see the [Red Dog Demo: Azure Container Apps Deployment](#) on GitHub.

The [Red Dog Demo: Microservices integration](#) is a packaged [app template](#) that builds on the preceding code assets to demonstrate the integration of Azure Container Apps, App Service, Functions, and API Management and provisions the infra, deploys the code using GitHub Actions.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Alice Gibbons](#) | Cloud Native Global Black Belt

Other contributors:

- [Kendall Roden](#) | Senior Program Manager
- [Lynn Orrell](#) | Principal Solution Specialist (GBB)

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Azure Container Apps docs](#)
- [Comparing container offerings in Azure](#)
- Other Reddog order management system implementations:
 - [Azure Arc hybrid deployment](#)
 - [AKS deployment](#)
 - [Local development](#)

Related resources

- [Microservices architecture on Azure Kubernetes Service](#)
- [Advanced Azure Kubernetes Service \(AKS\) microservices architecture](#)
- [CI/CD for AKS apps with Azure Pipelines](#)

Serverless functions architecture design

Article • 07/28/2022

Serverless architecture evolves cloud platforms toward pure cloud-native code by abstracting code from the infrastructure that it needs to run. [Azure Functions](#) is a serverless compute option that supports *functions*, small pieces of code that do single things.

Benefits of using serverless architectures with Functions applications include:

- The Azure infrastructure automatically provides all the updated servers that applications need to keep running at scale.
- Compute resources allocate dynamically, and instantly autoscale to meet elastic demands. Serverless doesn't mean "no server," but "less server," because servers run only as needed.
- Micro-billing saves costs by charging only for the compute resources and duration the code uses to execute.
- Function *bindings* streamline integration by providing declarative access to a wide variety of Azure and third-party services.

Functions are *event-driven*. An external event like an HTTP web request, message, schedule, or change in data *triggers* the function code. A Functions application doesn't code the trigger, only the response to the trigger. With a lower barrier to entry, developers can focus on business logic, rather than writing code to handle infrastructure concerns like messaging.

Azure Functions is a managed service in Azure and Azure Stack. The open source Functions runtime works in many environments, including Kubernetes, Azure IoT Edge, on-premises, and other clouds.

Serverless and Functions require new ways of thinking and new approaches to building applications. They aren't the right solutions for every problem. For example serverless Functions scenarios, see [Reference architectures](#).

Implementation steps

Successful implementation of serverless technologies with Azure Functions requires the following actions:

- Decide and plan

Architects and technical decision makers (TDMs) perform [application assessment](#), conduct or attend [technical workshops and trainings](#), run [proof of concept \(PoC\)](#) or [pilot](#) projects, and conduct architectural designs sessions as necessary.

- [Develop and deploy apps](#)

Developers implement serverless Functions app development patterns and practices, configure DevOps pipelines, and employ site reliability engineering (SRE) best practices.

- [Manage operations](#)

IT professionals identify hosting configurations, future-proof scalability by automating infrastructure provisioning, and maintain availability by planning for business continuity and disaster recovery.

- [Secure apps](#)

Security professionals handle Azure Functions security essentials, secure the hosting setup, and provide application security guidance.

Related resources

- To learn more about serverless technology, see the [Azure serverless documentation](#).
- To learn more about Azure Functions, see the [Azure Functions documentation](#).
- For help with choosing a compute technology, see [Choose an Azure compute service for your application](#).

Serverless Functions reference architectures

Article • 10/17/2023

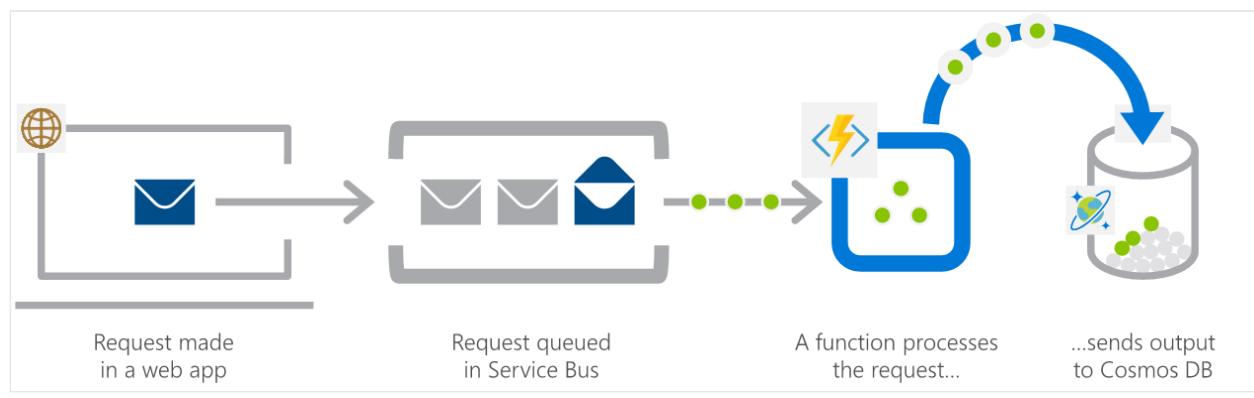
A reference architecture is a template of required components and the technical requirements to implement them. A reference architecture isn't custom-built for a customer solution, but is a high-level scenario based on extensive experience. Before designing a serverless solution, use a reference architecture to visualize an ideal technical architecture, then blend and integrate it into your environment.

Common serverless architecture patterns

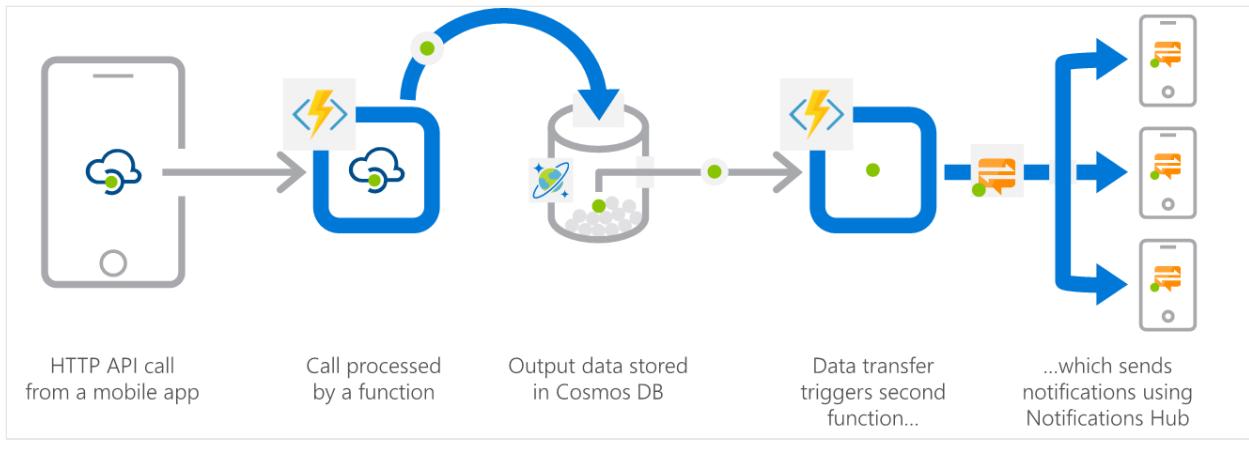
Common serverless architecture patterns include:

- Serverless APIs, mobile and web backends.
- Event and stream processing, Internet of Things (IoT) data processing, big data and machine learning pipelines.
- Integration and enterprise service bus to connect line-of-business systems, publish and subscribe (Pub/Sub) to business events.
- Automation and digital transformation and process automation.
- Middleware, software-as-a-Service (SaaS) like Dynamics, and big data projects.

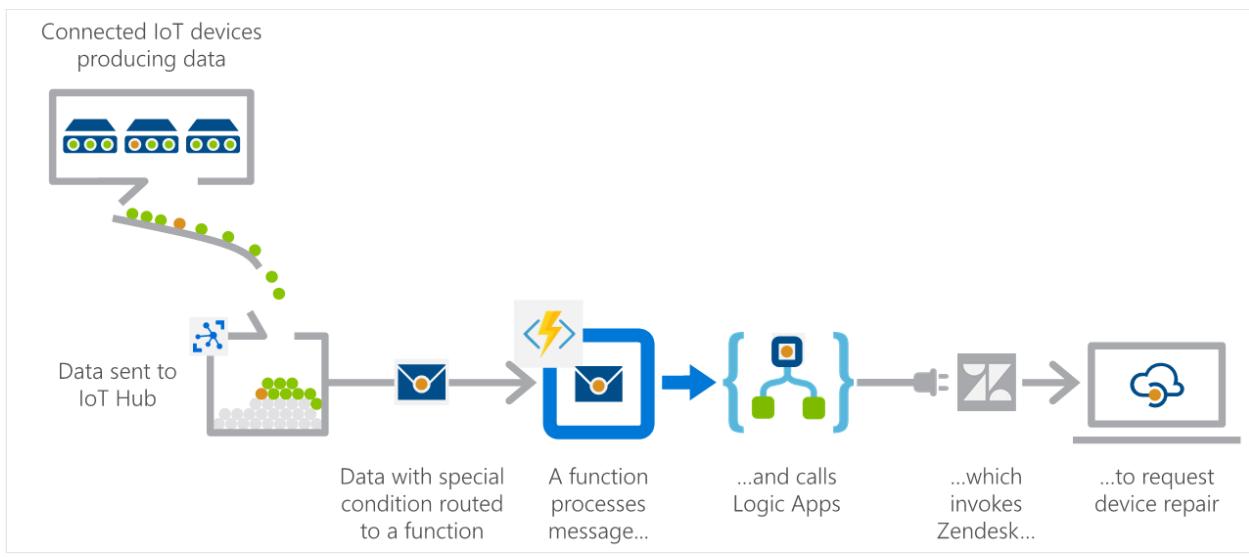
Web application backends the retail scenario: Pick up online orders from a queue, process them, and store the resulting data in a database.



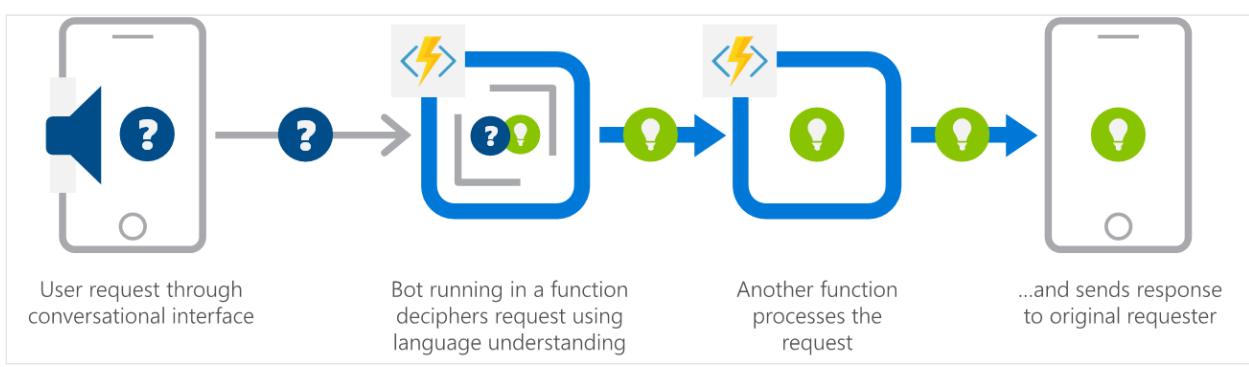
Mobile application backends the financial services scenario: Colleagues use mobile banking to reimburse each other for lunch. Whoever paid for lunch requests payment through a mobile app, which triggers a notification on the colleagues' phones.



IoT-connected backends in the manufacturing scenario: A manufacturing company uses IoT to monitor its machines. Functions detects anomalous data and that triggers a message to the service department when a repair is required.

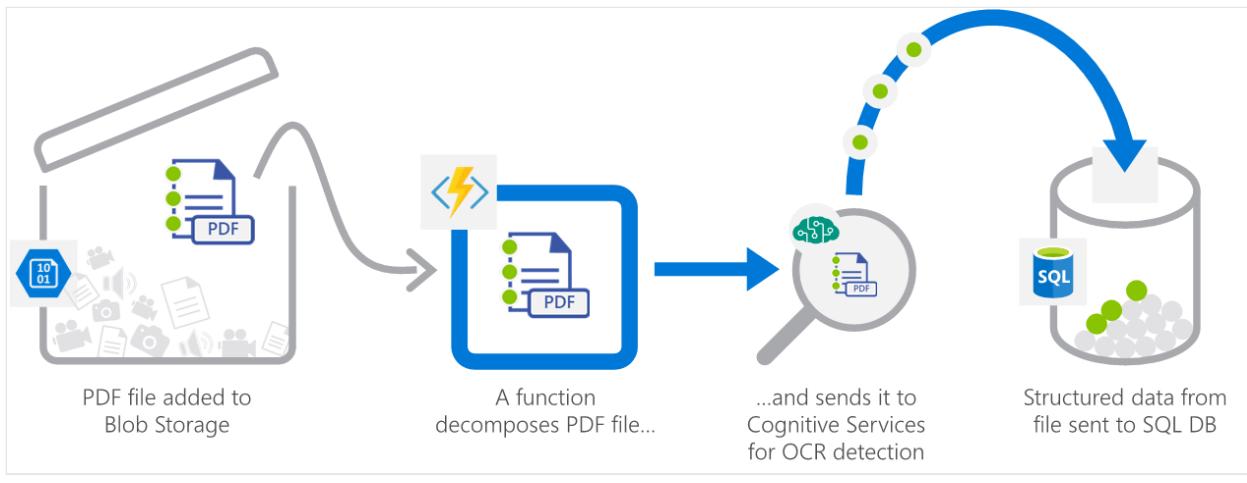


Conversational bot processing for the hospitality scenario: Customers ask for available vacation accommodations on their phones. A serverless bot deciphers requests and returns vacation options.

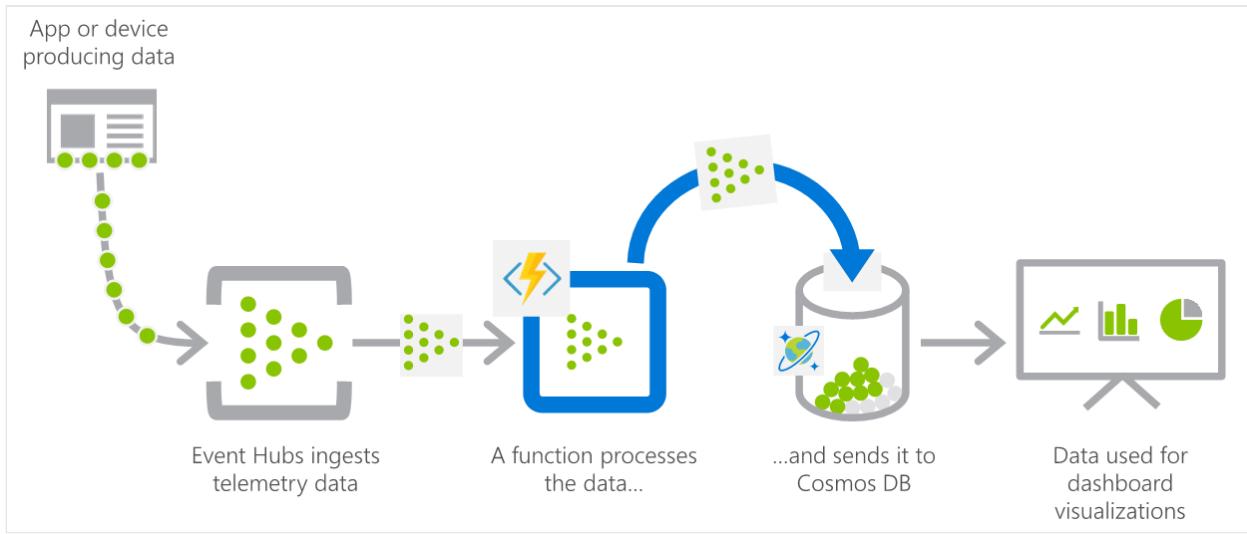


Real-time file processing for the healthcare scenario: The solution securely uploads patient records as PDF files. The solution then decomposes the data, by processes it

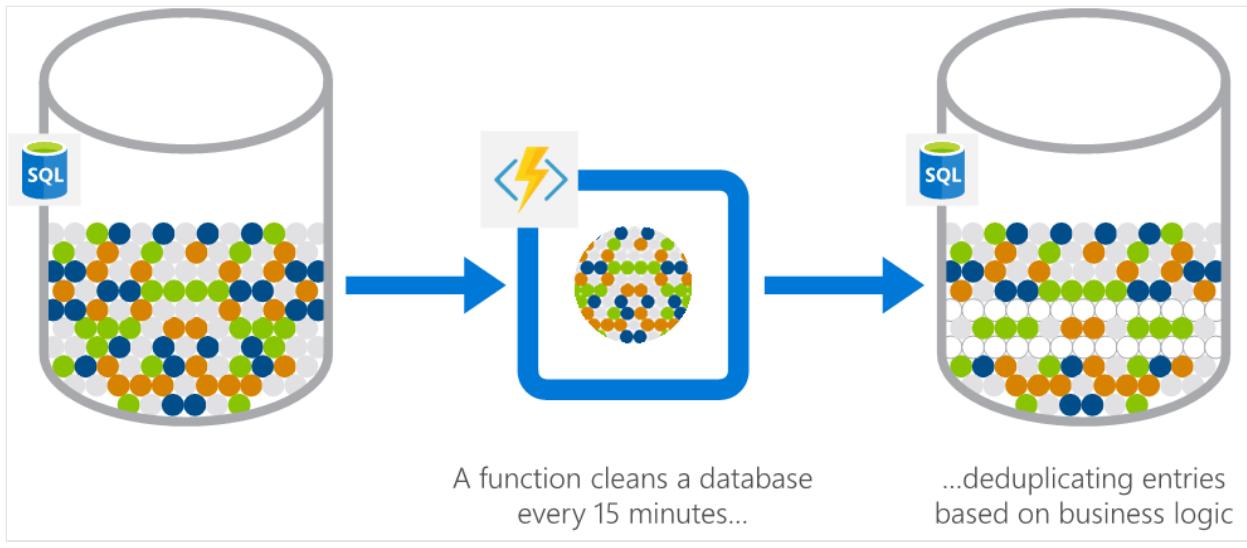
using OCR detection, and it adds the data to a database for easy queries.



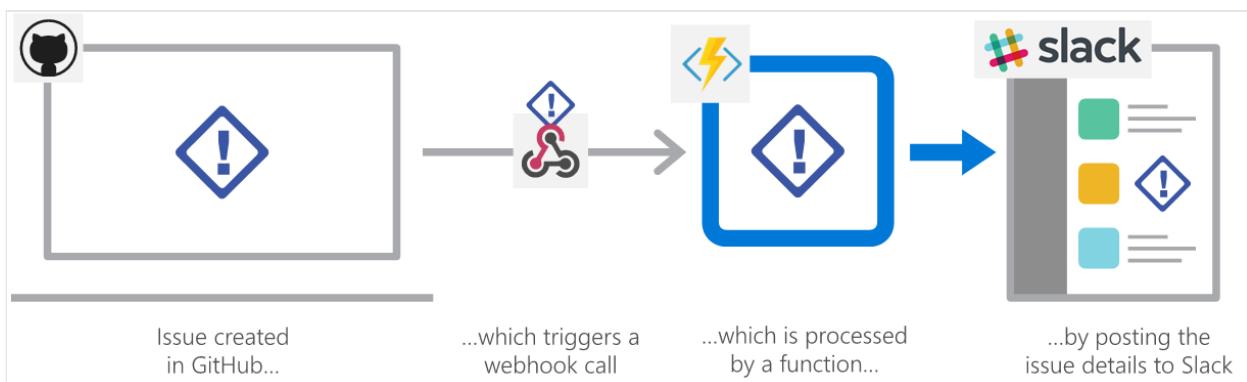
Real-time stream processing for the independent software vendor (ISV) scenario: A massive cloud app collects huge amounts of telemetry data. The app processes that data in near real-time and stores it in a database for use in an analytics dashboard.



Scheduled task automation for the financial services scenario: The app analyzes a customer database for duplicate entries every 15 minutes, to avoid sending out multiple communications to the same customers.



Extending SaaS applications in the professional services scenario: A SaaS solution provides extensibility through webhooks, which Azure Functions can implement to automate certain workflows.



Featured serverless reference architectures

The following featured serverless reference architectures walk through specific scenarios. See the linked articles for architectural diagrams and details.

Serverless microservices

The [serverless microservices reference architecture](#) walks you through designing, developing, and delivering the Rideshare application by Relecloud, a fictitious company. You get hands-on instructions for configuring and deploying all the architectural components, with helpful information about each component.

Serverless web application and event processing with Azure Functions

This two-part solution describes a hypothetical drone delivery system. Drones send in-flight status to the cloud, which stores these messages for later use. A web application allows users to retrieve the messages to get the latest device status.

- You can download the code for this solution from [GitHub](#)↗.
- The article [Code walkthrough: Serverless application with Azure Functions](#) walks you through the code and the design processes.

Event-based cloud automation

Automating workflows and repetitive tasks on the cloud can dramatically improve a DevOps team's productivity. A serverless model is best suited for event-driven automation scenarios. This [event-based automation reference architecture](#) illustrates two cloud automation scenarios: cost center tagging and throttling response.

Multicloud with Serverless Framework

The [Serverless Framework architecture](#) describes how the Microsoft Commercial Software Engineering (CSE) team partnered with a global retailer to deploy a highly-available serverless solution across both Azure and Amazon Web Services (AWS) cloud platforms, using the Serverless Framework.

More serverless Functions reference architectures

The following sections list other serverless and Azure Functions-related reference architectures and scenarios.

General

- [Serverless application architectures using Event Grid](#)
- [Serverless apps using Azure Cosmos DB](#)↗
- [Serverless event processing using Azure Functions](#)
- [Serverless web application on Azure](#)
- [Instant Broadcasting on Serverless Architecture](#)
- [Building a telehealth system on Azure](#)
- [Sharing location in real time using low-cost serverless Azure services](#)

Web and mobile backend

- An e-commerce front end
- Architect scalable e-commerce web app
- Baseline web application with zone redundancy
- Uploading and CDN-preloading static content with Azure Functions
- Cross Cloud Scaling Architecture
- Social App for Mobile and Web with Authentication

AI + Machine Learning

- Image classification for insurance claims
- Personalized Offers
- Personalized marketing solutions
- Speech transcription with Azure Cognitive Services
- Training a Model with AzureML and Azure Functions
- Customer Reviews App with Cognitive Services
- Enterprise-grade conversational bot
- AI at the Edge
- Mass ingestion and analysis of news feeds on Azure
- HIPAA and HITRUST compliant health data AI
- Intelligent Experiences On Containers (AKS, Functions, Keda) ↗

Data and analytics

- Application integration using Event Grid
- Mass ingestion and analysis of news feeds
- Tier Applications & Data for Analytics
- Operational analysis and driving process efficiency

IoT

- Azure IoT reference (SQL DB)
- Azure IoT reference (Azure Cosmos DB)
- IoT using Azure Cosmos DB
- Facilities management powered by mixed reality and IoT
- Complementary Code Pattern for Azure IoT Edge Modules & Cloud Applications ↗

Gaming

- Gaming using Azure Cosmos DB

Automation

- [Smart scaling for Azure Scale Set with Azure Functions](#)

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rogerio Halicki Cordeiro](#) ↗ | Senior Cloud Solution Architect

Plan for deploying serverless Functions

Article • 05/06/2022

To plan for moving an application to a serverless Azure Functions architecture, a technical decision maker (TDM) or architect:

- Verifies the application's characteristics and business requirements.
- Determines the application's suitability for serverless Azure Functions.
- Transforms business requirements into functional and other requirements.

Planning activities also include assessing technical team readiness, providing or attending workshops and training, and conducting architectural design reviews, proofs of concept, pilots, and technical implementations.

TDMs and architects may perform one or more of the following activities:

- **Execute an application assessment.** Evaluate the main aspects of the application to determine how complex and risky it is to rearchitect through application modernization, or rebuild a new cloud-native application. See [Application assessment](#).
- **Attend or promote technical workshops and training.** Host a Serverless workshop or CloudHack, or enjoy many other training and learning opportunities for serverless technologies, Azure Functions, app modernization, and cloud-native apps. See [Technical workshops and training](#).
- **Identify and execute a Proof of Concept (PoC) or pilot, or technical implementation.** Deliver a PoC, pilot, or technical implementation to provide evidence that serverless Azure Functions can solve a team's business problems. Showing teams how to modernize or build new cloud-native applications to their specifications can accelerate deployment to production. See [PoC or pilot](#).
- **Conduct architectural design sessions.** An architectural design session (ADS) is an in-depth discussion on how a new solution will blend into the environment. ADSs validate business requirements and transform them to functional and other requirements.

Next steps

For example scenarios that use serverless architectures with Azure Functions, see [Serverless reference architectures](#).

To move forward with serverless Azure Functions implementation, see the following resources:

- [Application development and deployment](#)
- [Azure Functions app operations](#)
- [Azure Functions app security](#)

Application assessment

Article • 06/16/2023

[Cloud rationalization](#) is the process of evaluating applications to determine the best way to migrate or modernize them for the cloud.

Rationalization methods include:

- **Rehost.** Also known as a *lift and shift* migration, rehost moves a current application to the cloud with minimal change.
- **Refactor.** Slightly refactoring an application to fit *platform-as-a-service* (PaaS)-based options can reduce operational costs.
- **Rearchitect.** Rearchitect aging applications that aren't compatible with cloud components, or cloud-compatible applications that would realize cost and operational efficiencies by rearchitecting into a cloud-native solution.
- **Rebuild.** If the changes or costs to carry an application forward are too great, consider creating a new cloud-native code base. Rebuild is especially appropriate for applications that previously met business needs, but are now unsupported or misaligned with current business processes.

Before you decide on an appropriate strategy, analyze the current application to determine the risk and complexity of each method. Consider application lifecycle, technology, infrastructure, performance, and operations and monitoring. For multilayer architectures, evaluate the presentation tier, service tier, integrations tier, and data tier.

The following checklists evaluate an application to determine the complexity and risk of rearchitecting or rebuilding.

Complexity and risk

Each of the following factors adds to complexity, risk, or both.

Architecture

Define the high-level architecture, such as web application, web services, data storage, or caching.

Factor	Complexity	Risk
Application components don't translate directly to Azure.	✓	✓
The application needs code changes to run in Azure.	✓	✓

Factor	Complexity	Risk
The application needs major, complex code changes to run in Azure.	✓	✓

Business drivers

Older applications might require extensive changes to get to the cloud.

Factor	Complexity	Risk
This application has been around for more than three years.	✓	
This application is business critical.		✓
There are technology blockers for migration.	✓	
There are business blockers for migration.		✓
This application has compliance requirements.		✓
The application is subject to data requirements that are specific to the country/region.		✓
The application is publicly accessible.	✓	✓

Technology

Factor	Complexity	Risk
This is not a web-based application, and isn't hosted on a web server.	✓	
The app isn't hosted in Windows IIS	✓	
The app isn't hosted on Linux	✓	
The application is hosted in a web farm, and requires multiple servers to host the web components.	✓	✓
The application requires third-party software to be installed on the servers.	✓	✓
The application is hosted in a single datacenter, and operations are performed in a single location.	✓	
The application accesses the server's registry.	✓	
The application sends emails, and needs access to an SMTP server.	✓	
This isn't a .NET application.	✓	

Factor	Complexity	Risk
The application uses SQL Server as its data store.	✓	
The application stores data on local disks, and needs access to the disks to operate properly.	✓	
The application uses Windows Services to process asynchronous operations, or needs external services to process data or operations.	✓	

Deployment

When assessing deployment requirements, consider:

- Number of daily users
- Average number of concurrent users
- Expected traffic
- Bandwidth in Gbps
- Requests per second
- Amount of memory needed

You can reduce deployment risk by storing code under source control in a version control system such as Git, Azure DevOps Server, or SVN.

Factor	Complexity	Risk
Using existing code and data is a #1 priority.	✓	✓
The application code isn't under source control.	✓	
There's no automated build process like Azure DevOps Server or Jenkins.	✓	
There's no automated release process to deploy the application.	✓	✓
The application has a Service Level Agreement (SLA) that dictates the amount of expected downtime.	✓	
The application experiences peak or variable usage times or loads.	✓	
The web application saves its session state in process, rather than an external data store.	✓	

Operations

Factor	Complexity	Risk

Factor	Complexity	Risk
The application doesn't have a well-established instrumentation strategy or standard instrumentation framework.	✓	
The application doesn't use monitoring tools, and the operations team doesn't monitor the app's performance.	✓	
The application has measured SLA in place, and the operations team monitors the application's performance.	✓	
The application writes to a log store, event log, log file, log database, or Application Insights.	✓	
The application doesn't write to a log store, event log, log file, log database, or Application Insights.	✓	
The application isn't part of the organization's disaster recovery plan.	✓	

Security

Factor	Complexity	Risk
The application uses Active Directory to authenticate users.	✓	✓
The organization hasn't yet configured Azure Active Directory (Azure AD), or hasn't configured Azure AD Connect to synchronize on-premises AD with Azure AD.	✓	
The application requires access to on-premises resources, which will require VPN connectivity from Azure.	✓	
The organization hasn't yet configured a VPN connection between Azure and their on-premises environment.	✓	✓
The application requires an SSL certificate to run.	✓	✓

Results

Using the tables above, determine if each factor applies to your application. Count the number of **Complexity** and **Risk** checkmarks for the factors that apply to your application.

- The expected level of complexity to migrate or modernize the application to Azure is: **Matching Complexity Factors/Total Possible Complexity Factors**.
- The expected risk involved is: **Matching Risk Factors/Total Possible Risk Factors**.

Total Possible Complexity Factors = 28, Total Possible Risk Factors = 23

For both complexity and risk, a score obtained from the calculation above of <0.3 = low, <0.7 = medium, >0.7 = high. These scores provide a relative scale of complexity and risk.

Refactor, rearchitect, or rebuild

To rationalize whether to rehost, refactor, rearchitect, or rebuild your application, consider the following points. Many of these factors also contribute to complexity and risk.

Determine whether the application components can translate directly to Azure. If so, you don't need code changes to move the application to Azure, and could use rehost or refactor strategies. If not, you need to rewrite code, so you need to rearchitect or rebuild.

If the app does need code changes, determine the complexity and extent of the needed changes. Minor changes might allow for rearchitecting, while major changes may require rebuilding.

Rehost or refactor

- If using existing code and data is a top priority, consider a refactor strategy rather than rearchitecting or rebuilding.
- If you have pressing timelines like datacenter shutdown or contract expiration, end-of-life licensing, or mergers or acquisitions, the fastest way to get the application to Azure might be to rehost, followed by refactoring to take advantage of cloud capabilities.

Rearchitect or rebuild

- If there are applications serving similar needs in your portfolio, this might be an opportunity to rearchitect or rebuild the entire solution.
- If you want to [implement multi-tier or microservices architecture](#) for a monolithic app, you must rearchitect or rebuild the app. If you don't mind retaining the monolithic structure, you might be able to rehost or refactor.
- Rearchitect or rebuild the app to take advantage of cloud capabilities if you plan to update the app more often than yearly, if the app has peak or variable usage

times, or if you expect the app to handle high traffic.

To decide between rearchitecting or rebuilding, assess the following factors. The largest scoring result indicates your best strategy.

Factor	Rearchitect	Rebuild
There are other applications serving similar needs in your portfolio.	✓	✓
The application needs minor code changes to run in Azure.	✓	
The application needs major, complex code changes to run in Azure.		✓
It's important to use existing code.	✓	
You want to move a monolithic application to multi-tier architecture.	✓	
You want to move a monolithic application to a microservices architecture.	✓	✓
You expect this app to add breakthrough capabilities like AI, IoT, or bots.		✓
Among functionality, cost, infrastructure, and processes, functionality is the least efficient aspect of this application.		✓
The application requires third-party software installed on the servers.	✓	
The application accesses the server's registry.	✓	
The application sends emails and needs access to an SMTP server.	✓	
The application uses SQL Server as its data store.	✓	
The application stores data on local disks, and needs access to the disks to run properly.	✓	
The application uses Windows services to process asynchronous operations, or needs external services to process data or operations.	✓	
A web application saves its session state in process, rather than to an external data store.	✓	
The app has peak and variable usage times and loads.	✓	✓
You expect the application to handle high traffic.	✓	✓

Next steps

- [What is a digital estate?](#)
- [Approaches to digital estate planning](#)

- Rationalize the digital estate

Related resources

- Migration architecture design
- Build migration plan with Azure Migrate

Technical workshops and training

Article • 01/03/2024

The workshops, classes, and learning materials in this article provide technical training for serverless architectures with Azure Functions. These resources help you and your team or customers understand and implement application modernization and cloud-native apps.

Technical workshops

The [Microsoft Cloud Workshop \(MCW\)](#) program provides workshops you can host to foster cloud learning and adoption. Each workshop includes presentation decks, trainer and student guides, and hands-on lab guides. Contribute your own content and feedback to add to a robust database of training guides for deploying advanced Azure workloads on the Microsoft Cloud Platform.

Workshops related to application development workloads include:

- [Serverless APIs in Azure](#). Set of entry-level exercises, which cover the basics of building and managing serverless APIs in Microsoft Azure - with Azure Functions, Azure API Management, and Azure Application Insights.

Instructor-led training

[Course AZ-204: Developing solutions for Microsoft Azure](#) teaches developers how to create end-to-end solutions in Microsoft Azure. Students learn how to implement Azure compute solutions, create Azure Functions, implement and manage web apps, develop solutions utilizing Azure Storage, implement authentication and authorization, and secure their solutions by using Azure Key Vault and managed identities. Students also learn to connect to and consume Azure and third-party services, and include event- and message-based models in their solutions. The course also covers monitoring, troubleshooting, and optimizing Azure solutions.

Serverless OpenHack

The Serverless [OpenHack](#) simulates a real-world scenario where a company wants to utilize serverless services to build and release an API to integrate into their distributor's application. This OpenHack lets attendees quickly build and deploy Azure serverless solutions with cutting-edge compute services like Azure Functions, Logic Apps, Event

Grid, Service Bus, Event Hubs, and Azure Cosmos DB. The OpenHack also covers related technologies like API Management, Azure DevOps or GitHub, Application Insights, Dynamics 365/Microsoft 365, and Cognitive APIs.

OpenHack attendees build serverless functions, web APIs, and a CI/CD pipeline to support them. They implement further serverless technologies to integrate line of business (LOB) app workflows, process user and data telemetry, and create key progress indicator (KPI)-aligned reports. By the end of the OpenHack, attendees have built out a full serverless technical solution that can create workflows between systems and handle events, files, and data ingestion.

Microsoft customer projects inspired these OpenHack challenges:

- Configure the developer environment.
- Create your first serverless function and workflow.
- Build APIs to support business needs.
- Deploy a management layer for APIs and monitoring usage.
- Build a LOB workflow process.
- Process large amounts of unstructured file data.
- Process large amounts of incoming event data.
- Implement a publisher/subscriber messaging pattern and virtual network integration.
- Conduct sentiment analysis.
- Perform data aggregation, analysis, and reporting.

To attend an OpenHack, register at <https://openhack.microsoft.com>. For enterprises with many engineers, Microsoft can request and organize a dedicated Serverless OpenHack.

Microsoft Learn

Microsoft Learn is a free online platform that provides interactive learning for Microsoft products. The goal is to improve proficiency with fun, guided, hands-on content that's specific to your role and goals. Learning paths are collections of modules that are organized around specific roles like developer, architect, or system admin, or technologies like Azure Web Apps, Azure Functions, or Azure SQL DB. Learning paths provide understanding of different aspects of the technology or role.

Learning paths about serverless apps and Azure Functions include:

- [Create serverless applications](#). Learn how to leverage functions to execute server-side logic and build serverless architectures.

- [Architect message brokering and serverless applications in Azure](#). Learn how to create reliable messaging for your applications, and how to take advantage of serverless application services in Azure.
- [Search all Functions-related learning paths](#).

Hands-on labs and how-to guides

- [Build a serverless web app](#), from the Build 2018 conference
- [Build a Serverless IoT Solution with Python Azure Functions and SignalR](#) ↗

Next steps

- [Execute an application assessment](#)
- [Identify and execute a PoC or Pilot project](#)

Proof of concept or pilot

Article • 05/06/2022

When driving a technical and security decision for your company or customer, a *Proof of Concept (PoC)* or *pilot* is an opportunity to deliver evidence that the proposed solution solves the business problems. The PoC or pilot increases the likelihood of a successful adoption.

A PoC:

- Demonstrates that a business model or idea is feasible and will work to solve the business problem
- Usually involves one to three features or capabilities
- Can be in one or multiple technologies
- Is geared toward a particular scenario, and proves what the customer needs to know to make the technical or security decision
- Is used only as a demonstration and won't go into production
- Is IT-driven and enablement-driven

A pilot:

- Is a test run or trial of a proposed action or product
- Lasts longer than a PoC, often weeks or months
- Has a higher return on investment (ROI) than a PoC
- Builds in a pre-production or trial environment, with the intent that it will then go into production
- Is adoption-driven and consumption-driven

PoC and pilot best practices

Be aware of compliance issues when working in a customer's environment, and make sure your actions are always legal and compliant.

- Touching or altering the customer's environment usually requires a contract, and may involve a partner or Microsoft services. Without a contract, your company may be liable for issues or damages.
- Governance may require Legal department approval. Your company may not be able to give away intellectual property (IP) for free. You may need a legal contract or contracts to specify whether your company or the customer pays for the IP.
- Get disclosure guidance when dealing with non-disclosure agreements (NDAs), product roadmaps, NDA features, or anything not released to the general public.

- In a pilot, don't use a trial Microsoft Developers Network (MSDN) environment, or any environment that you own.
- Use properly licensed software, and ask the opportunity owner to make sure to handle software licensing correctly.

The customer, partner, or your company may pay for the PoC or pilot. Depending on the size of the contract, the ROI, and the cost of sale, one group may cover it all, or a combination of all three parties may cover the cost. Ensure that your company or customer has some investment in the PoC or pilot. If they don't, this can be a red flag signaling that your company or customer doesn't yet see value in the solution.

PoC and pilot process

The Technical Decision Maker (TDM) is responsible for driving an adoption decision. The TDM is responsible for ensuring that the right partners and resources are involved in a PoC or pilot. As a TDM, make sure you're aware of the partners in your product and service area or region. Be aware of their key service offerings around your product service area.

Planning

Consider the following health questions:

- Do you have a good technical plan, including key decision makers and Microsoft potential?
- Can you deliver the needed assurance without a PoC?
- Should you switch to a pilot?
- What are the detailed scope and decision criteria your team or customer agreed to?
- If you meet the criteria, will your company or customer buy or deploy the solution?

Do the following tasks:

- Analyze risk.
- Evaluate the setting.
- Do the preparation.
- Consider workloads and human resources.
- Present PoC or pilot health status.
- Fulfill technical prerequisites.
- Define the go/no go decision.
- Create a final project plan specification.

Execution

For the execution phase:

- Determine who kicks off the presentation.
- Schedule the meeting in the morning, if possible.
- Prepare the demos and slides.
- Conduct a dry run to refine the presentation.
- Get feedback.
- Involve your company or customer team.
- Complete the win/lose statement.

Debriefing

During the debriefing phase, consider:

- Whether criteria were met or not met
- Stakeholder investment
- Initiating deployment
- Finding a partner and training
- Lessons learned
- Corrections or extensions of PoC or pilot guidance
- Archiving of valuable deliverables

Change management

Change management uses tested methods and techniques to avoid errors and minimize impact when administering change.

Ideally, a pilot includes a cross-section of users, to address any potential issues or problems that arise. Users may be comfortable and familiar with their old technology, and have difficulty moving into new technical solutions. Change management keeps this in mind, and helps the user understand the reasons behind the change and the impact the change will make.

This understanding is part of a pilot, and addresses everyone who has a stake in the project. A pilot is better than a PoC, because the customer is more involved, so they're more likely to implement the change.

The pilot includes a detailed follow-up through surveys or focus groups. The feedback can prove and improve the change.

Next steps

- Execute an application assessment
- Promote a technical workshop or training
- Code a technical implementation with the team or customer

Related resources

[Prosci® change management training ↗](#)

Application development and deployment

Article • 06/30/2023

To develop and deploy serverless applications with Azure Functions, examine patterns and practices, configure DevOps pipelines, and implement site reliability engineering (SRE) best practices.

For detailed information about serverless architectures and Azure Functions, see:

- [Serverless apps: Architecture, patterns, and Azure implementation](#)
- [Azure Serverless Computing Cookbook ↗](#)
- [Example serverless reference architectures](#)

Planning

To plan app development and deployment:

1. Prepare development environment and set up workflow.
2. Structure projects to support Azure Functions app development.
3. Identify app triggers, bindings, and configuration requirements.

Understand event-driven architecture

A different event triggers every function in a serverless Functions project. For more information about event-driven architectures, see:

- [Event-driven architecture style.](#)
- [Event-driven design patterns to enhance existing applications using Azure Functions](#)

Prepare development environment

Set up your development workflow and environment with the tools to create Functions. For details about development tools and Functions code project structure, see:

- [Code and test Azure Functions locally](#)
- [Develop Azure Functions by using Visual Studio Code](#)
- [Develop Azure Functions using Visual Studio](#)
- [Work with Azure Functions Core Tools](#)

- [Folder structure](#)

Development

Decide on the development language to use. Azure Functions supports C#, F#, PowerShell, JavaScript, TypeScript, Java, and Python. All of a project's Functions must be in the same language. For more information, see [Supported languages in Azure Functions](#).

Define triggers and bindings

A trigger invokes a Function, and every Function must have exactly one trigger. Binding to a Function declaratively connects another resource to the Function. For more information about Functions triggers and bindings, see:

- [Azure Functions triggers and bindings concepts](#)
- [Execute an Azure Function with triggers](#)
- [Chain Azure Functions together using input and output bindings](#)

Create the Functions application

Functions follow the single responsibility principle: do only one thing. For more information about Functions development, see:

- [Azure Functions developers guide](#)
- [Create serverless applications](#)
- [Strategies for testing your code in Azure Functions](#)
- [Functions best practices](#)

Use Durable Functions for stateful workflows

Durable Functions in Azure Functions let you define stateful workflows in a serverless environment by writing *orchestrator functions*, and stateful entities by writing *entity functions*. Durable Functions manage state, checkpoints, and restarts, allowing you to focus on business logic. For more information, see [What are Durable Functions](#).

Understand and address cold starts

If the number of serverless host instances scales down to zero, the next request has the added latency of restarting the Function app, called a *cold start*. To minimize the performance impact of cold starts, reduce dependencies that the Functions app needs

to load on startup, and use as few large, synchronous calls and operations as possible. For more information about autoscaling and cold starts, see [Serverless Functions operations](#).

Manage application secrets

For security, don't store credentials in application code. To use Azure Key Vault with Azure Functions to store and retrieve keys and credentials, see [Use Key Vault references for App Service and Azure Functions](#).

For more information about serverless Functions application security, see [Serverless Functions security](#).

Deployment

To prepare serverless Functions application for production, make sure you can:

- Fulfill application resource requirements.
- Monitor all aspects of the application.
- Diagnose and troubleshoot application issues.
- Deploy new application versions without affecting production systems.

Define deployment technology

Decide on deployment technology, and organize scheduled releases. For more information about how Functions app deployment enables reliable, zero-downtime upgrades, see [Deployment technologies in Azure Functions](#).

Avoid using too many resource connections

Functions in a Functions app share resources, including connections to HTTPS, databases, and services such as Azure Storage. When many Functions are running concurrently, it's possible to run out of available connections. For more information, see [Manage connections in Azure Functions](#).

Configure logging, alerting, and application monitoring

Application Insights in Azure Monitor collects log, performance, and error data. Application Insights automatically detects performance anomalies, and includes powerful analytics tools to help diagnose issues and understand function usage.

For more information about application monitoring and logging, see:

- [Monitor Azure Functions](#)
- [Monitoring Azure Functions with Azure Monitor Logs](#)
- [Application Insights for Azure Functions supported features](#)

Diagnose and troubleshoot issues

Learn how to effectively use diagnostics for troubleshooting in proactive and problem-first scenarios. For more information, see:

- [Keep your Azure App Service and Azure Functions apps healthy and happy ↗](#)
- [Troubleshoot error: "Azure Functions Runtime is unreachable"](#)

Deploy applications using an automated pipeline and DevOps

Full automation of all steps from code commit to production deployment lets teams focus on building code, and removes the overhead and potential human error of manual steps. Deploying new code is quicker and less risky, helping teams become more agile, more productive, and more confident about their code.

For more information about DevOps and continuous deployment (CD), see:

- [Continuous deployment for Azure Functions](#)
- [Continuous delivery by using Azure DevOps](#)
- [Continuous delivery by using GitHub Action](#)

Optimization

Once the application is in production, prepare for scaling and implement site reliability engineering (SRE).

Ensure optimal scalability

For information about factors that impact Functions app scalability, see:

- [Scalability best practices](#)
- [Performance and scale in Durable Functions](#)

Implement SRE practices

Site Reliability Engineering (SRE) is a proven approach to maintaining crucial system and application reliability, while iterating at the speed the marketplace demands. For more information, see:

- [Introduction to Site Reliability Engineering \(SRE\)](#)
- [DevOps at Microsoft: Game streaming SRE ↗](#)

Next steps

For hands-on serverless Functions app development and deployment walkthroughs, see:

- [Serverless Functions code walkthrough](#)
- [CI/CD for a serverless frontend](#)

For an engineering playbook to help teams and customers successfully implement serverless Functions projects, see the [Code-With Customer/Partner Engineering Playbook ↗](#).

Code walkthrough: Serverless application with Functions

Azure Event Hubs

Azure Functions

Serverless models abstract code from the underlying compute infrastructure, allowing developers to focus on business logic without extensive setup. Serverless code reduces costs, because you pay only for the code execution resources and duration.

The serverless event-driven model fits situations where a certain event triggers a defined action. For example, receiving an incoming device message triggers storage for later use, or a database update triggers some further processing.

To help you explore Azure serverless technologies in Azure, Microsoft developed and tested a serverless application that uses [Azure Functions](#). This article walks through the code for the serverless Functions solution, and describes design decisions, implementation details, and some of the "gotchas" you might encounter.

Explore the solution

The two-part solution describes a hypothetical drone delivery system. Drones send in-flight status to the cloud, which stores these messages for later use. A web app lets users retrieve the messages to get the latest status of the devices.

You can download the code for this solution from [GitHub](#).

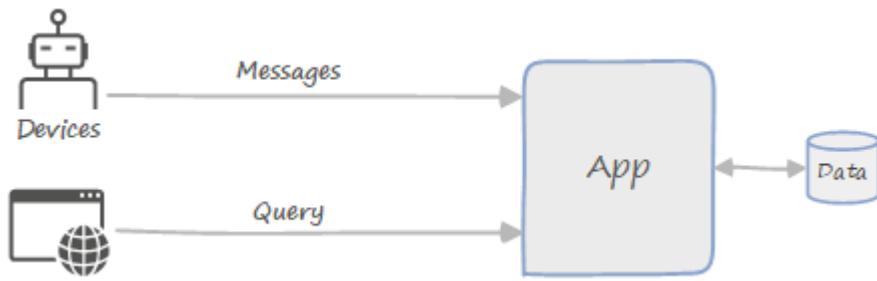
This walkthrough assumes basic familiarity with the following technologies:

- [Azure Functions](#)
- [Azure Event Hubs](#)
- [.NET Core](#)

You don't need to be an expert in Functions or Event Hubs, but you should understand their features at a high level. Here are some good resources to get started:

- [An introduction to Azure Functions](#)
- [Features and terminology in Azure Event Hubs](#)

Understand the scenario



Fabrikam manages a fleet of drones for a drone delivery service. The application consists of two main functional areas:

- **Event ingestion.** During flight, drones send status messages to a cloud endpoint. The application ingests and processes these messages, and writes the results to a back-end database (Azure Cosmos DB). The devices send messages in [protocol buffer](#) (protobuf) format. Protobuf is an efficient, self-describing serialization format.

These messages contain partial updates. At a fixed interval, each drone sends a "key frame" message that contains all of the status fields. Between key frames, the status messages only include fields that changed since the last message. This behavior is typical of many IoT devices that need to conserve bandwidth and power.

- **Web app.** A web application allows users to look up a device and query the device's last-known status. Users must sign into the application and authenticate with Microsoft Entra ID. The application only allows requests from users who are authorized to access the app.

Here's a screenshot of the web app, showing the result of a query:

Fabrikam

This sample demonstrates how to authenticate a serverless API

Get status	drone-61
Property	Value
id	drone-61
Battery	1
FlightMode	5
Latitude	47.476075
Longitude	-122.192026
Altitude	0
GyrometerOK	true
AccelerometerOK	true
MagnetometerOK	true

Design the application

Fabrikam has decided to use Azure Functions to implement the application business logic. Azure Functions is an example of "Functions as a Service" (FaaS). In this computing model, a *function* is a piece of code that is deployed to the cloud and runs in a hosting environment. This hosting environment completely abstracts the servers that run the code.

Why choose a serverless approach?

A serverless architecture with Functions is an example of an event-driven architecture. The function code is triggered by some event that's external to the function — in this case, either a message from a drone, or an HTTP request from a client application. With a function app, you don't need to write any code for the trigger. You only write the code that runs in response to the trigger. That means you can focus on your business logic, rather than writing a lot of code to handle infrastructure concerns like messaging.

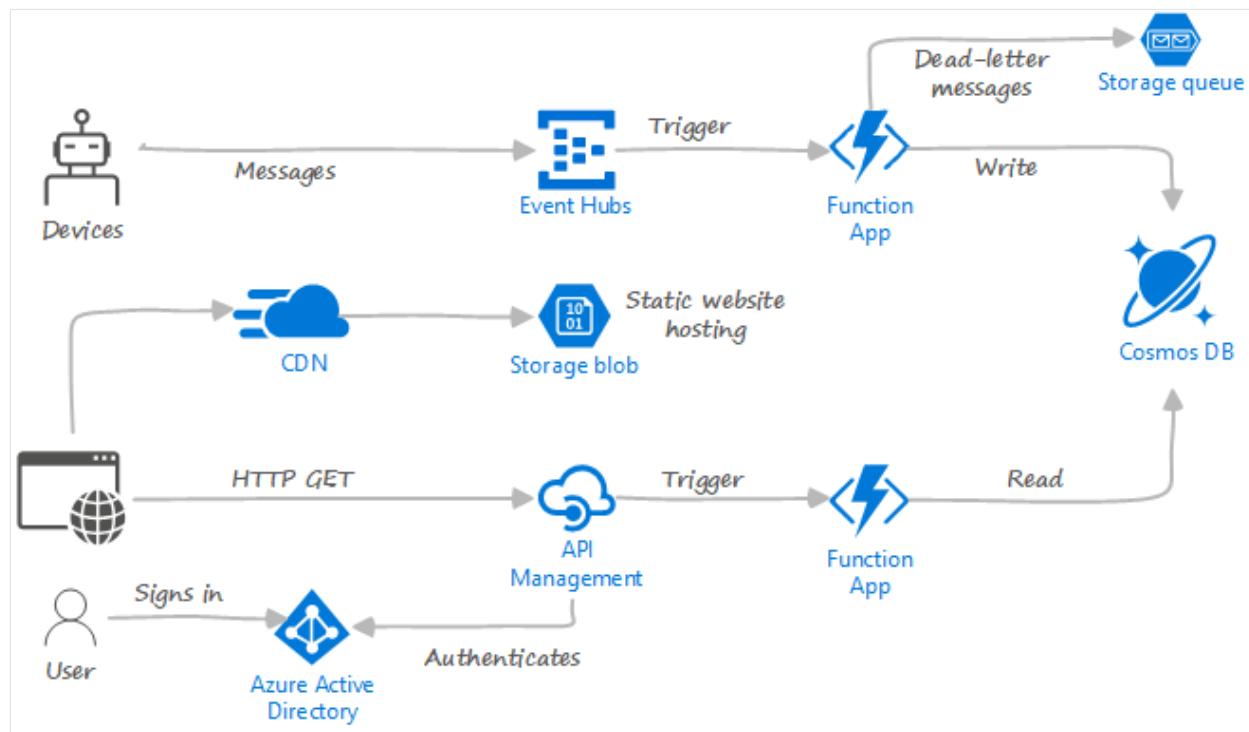
There are also some operational advantages to using a serverless architecture:

- There is no need to manage servers.

- Compute resources are allocated dynamically as needed.
- You are charged only for the compute resources used to execute your code.
- The compute resources scale on demand based on traffic.

Architecture

The following diagram shows the high-level architecture of the application:



Event ingestion:

1. Drone messages are ingested by Azure Event Hubs.
2. Event Hubs produces a stream of events that contain the message data.
3. These events trigger an Azure Functions app to process them.
4. The results are stored in Azure Cosmos DB.

Web app:

1. Static files are served by CDN from Blob storage.
2. A user signs into the web app using Microsoft Entra ID.
3. Azure API Management acts as a gateway that exposes a REST API endpoint.
4. HTTP requests from the client trigger an Azure Functions app that reads from Azure Cosmos DB and returns the result.

This application is based on two reference architectures, corresponding to the two functional blocks described above:

- [Serverless event processing using Azure Functions](#)
- [Serverless web application on Azure](#)

You can read those articles to learn more about the high-level architecture, the Azure services that are used in the solution, and considerations for scalability, security, and reliability.

Drone telemetry function

Let's start by looking at the function that processes drone messages from Event Hubs. The function is defined in a class named `RawTelemetryFunction`:

C#

```
namespace DroneTelemetryFunctionApp
{
    public class RawTelemetryFunction
    {
        private readonly ITelemetryProcessor telemetryProcessor;
        private readonly IStateChangeProcessor stateChangeProcessor;
        private readonly TelemetryClient telemetryClient;

        public RawTelemetryFunction(ITelemetryProcessor telemetryProcessor,
IStateChangeProcessor stateChangeProcessor, TelemetryClient telemetryClient)
        {
            this.telemetryProcessor = telemetryProcessor;
            this.stateChangeProcessor = stateChangeProcessor;
            this.telemetryClient = telemetryClient;
        }
    }
    ...
}
```

This class has several dependencies, which are injected into the constructor using dependency injection:

- The `ITelemetryProcessor` and `IStateChangeProcessor` interfaces define two helper objects. As we'll see, these objects do most of the work.
- The `TelemetryClient` is part of the Application Insights SDK. It is used to send custom application metrics to Application Insights.

Later, we'll look at how to configure the dependency injection. For now, just assume these dependencies exist.

Configure the Event Hubs trigger

The logic in the function is implemented as an asynchronous method named `RunAsync`.

Here is the method signature:

C#

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection",
    ConsumerGroup = "%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage>
    deadLetterMessages,
    ILogger logger)
{
    // implementation goes here
}
```

The method takes the following parameters:

- `messages` is an array of event hub messages.
- `deadLetterMessages` is an Azure Storage Queue, used for storing dead letter messages.
- `logging` provides a logging interface, for writing application logs. These logs are sent to Azure Monitor.

The `EventHubTrigger` attribute on the `messages` parameter configures the trigger. The properties of the attribute specify an event hub name, a connection string, and a [consumer group](#). (A *consumer group* is an isolated view of the Event Hubs event stream. This abstraction allows for multiple consumers of the same event hub.)

Notice the percent signs (%) in some of the attribute properties. These indicate that the property specifies the name of an app setting, and the actual value is taken from that app setting at run time. Otherwise, without percent signs, the property gives the literal value.

The `Connection` property is an exception. This property always specifies an app setting name, never a literal value, so the percent sign is not needed. The reason for this distinction is that a connection string is secret and should never be checked into source code.

While the other two properties (event hub name and consumer group) are not sensitive data like a connection string, it's still better to put them into app settings, rather than hard coding. That way, they can be updated without recompiling the app.

For more information about configuring this trigger, see [Azure Event Hubs bindings for Azure Functions](#).

Message processing logic

Here's the implementation of the `RawTelemetryFunction.RunAsync` method that processes a batch of messages:

C#

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection",
    ConsumerGroup = "%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage>
deadLetterMessages,
    ILogger logger)
{
    telemetryClient.GetMetric("EventHubMessageBatchSize").TrackValue(messages.Length);

    foreach (var message in messages)
    {
        DeviceState deviceState = null;

        try
        {
            deviceState = telemetryProcessor.Deserialize(message.Body.Array,
logger);

            try
            {
                await stateChangeProcessor.UpdateState(deviceState, logger);
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error updating status document",
deviceState);
                await deadLetterMessages.AddAsync(new DeadLetterMessage {
Exception = ex, EventData = message, DeviceState = deviceState });
            }
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error deserializing message",
message.SystemProperties.PartitionKey,
message.SystemProperties.SequenceNumber);
            await deadLetterMessages.AddAsync(new DeadLetterMessage {
Exception = ex, EventData = message });
        }
    }
}
```

```
        }  
    }  
}
```

When the function is invoked, the `messages` parameter contains an array of messages from the event hub. Processing messages in batches will generally yield better performance than reading one message at a time. However, you have to make sure the function is resilient and handles failures and exceptions gracefully. Otherwise, if the function throws an unhandled exception in the middle of a batch, you might lose the remaining messages. This consideration is discussed in more detail in the section [Error handling](#).

But if you ignore the exception handling, the processing logic for each message is simple:

1. Call `ITelemetryProcessor.Deserialize` to deserialize the message that contains a device state change.
2. Call `IStateChangeProcessor.UpdateState` to process the state change.

Let's look at these two methods in more detail, starting with the `Deserialize` method.

Deserialize method

The `TelemetryProcessor.Deserialize` method takes a byte array that contains the message payload. It deserializes this payload and returns a `DeviceState` object, which represents the state of a drone. The state may represent a partial update, containing just the delta from the last-known state. Therefore, the method needs to handle `null` fields in the serialized payload.

C#

```
public class TelemetryProcessor : ITelemetryProcessor  
{  
    private readonly ITelemetrySerializer<DroneState> serializer;  
  
    public TelemetryProcessor(ITelemetrySerializer<DroneState> serializer)  
    {  
        this.serializer = serializer;  
    }  
  
    public DeviceState Deserialize(byte[] payload, ILogger log)  
    {  
        DroneState restored = serializer.Deserialize(payload);  
  
        log.LogInformation("Deserialize message for device ID {DeviceId}",  
restored.DeviceId);
```

```

    var deviceState = new DeviceState();
    deviceState.DeviceId = restored.DeviceId;

    if (restored.Battery != null)
    {
        deviceState.Battery = restored.Battery;
    }
    if (restored.FlightMode != null)
    {
        deviceState.FlightMode = (int)restored.FlightMode;
    }
    if (restored.Position != null)
    {
        deviceState.Latitude = restored.Position.Value.Latitude;
        deviceState.Longitude = restored.Position.Value.Longitude;
        deviceState.Altitude = restored.Position.Value.Altitude;
    }
    if (restored.Health != null)
    {
        deviceState.AccelerometerOK =
restored.Health.Value.AccelerometerOK;
        deviceState.GyrometerOK = restored.Health.Value.GyrometerOK;
        deviceState.MagnetometerOK =
restored.Health.Value.MagnetometerOK;
    }
    return deviceState;
}
}

```

This method uses another helper interface, `ITelemetrySerializer<T>`, to deserialize the raw message. The results are then transformed into a [POCO](#) model that is easier to work with. This design helps to isolate the processing logic from the serialization implementation details. The `ITelemetrySerializer<T>` interface is defined in a shared library, which is also used by the device simulator to generate simulated device events and send them to Event Hubs.

C#

```

using System;

namespace Serverless.Serialization
{
    public interface ITelemetrySerializer<T>
    {
        T Deserialize(byte[] message);

        ArraySegment<byte> Serialize(T message);
    }
}

```

UpdateState method

The `StateChangeProcessor.UpdateState` method applies the state changes. The last-known state for each drone is stored as a JSON document in Azure Cosmos DB. Because the drones send partial updates, the application can't simply overwrite the document when it gets an update. Instead, it needs to fetch the previous state, merge the fields, and then perform an upsert operation.

C#

```
public class StateChangeProcessor : IStateChangeProcessor
{
    private IDocumentClient client;
    private readonly string cosmosDBDatabase;
    private readonly string cosmosDBCcollection;

    public StateChangeProcessor(IDocumentClient client,
        IOptions<StateChangeProcessorOptions> options)
    {
        this.client = client;
        this.cosmosDBDatabase = options.Value.COSMOSDB_DATABASE_NAME;
        this.cosmosDBCcollection = options.Value.COSMOSDB_DATABASE_COL;
    }

    public async Task<ResourceResponse<Document>> UpdateState(DeviceState
        source, ILogger log)
    {
        log.LogInformation("Processing change message for device ID
        {DeviceId}", source.DeviceId);

        DeviceState target = null;

        try
        {
            var response = await
                client.ReadDocumentAsync(UriFactory.CreateDocumentUri(cosmosDBDatabase,
                    cosmosDBCcollection, source.DeviceId),
                new
                    RequestOptions { PartitionKey = new PartitionKey(source.DeviceId) });

            target = (DeviceState)(dynamic)response.Resource;

            // Merge properties
            target.Battery = source.Battery ?? target.Battery;
            target.FlightMode = source.FlightMode ?? target.FlightMode;
            target.Latitude = source.Latitude ?? target.Latitude;
            target.Longitude = source.Longitude ?? target.Longitude;
            target.Altitude = source.Altitude ?? target.Altitude;
            target.AccelerometerOK = source.AccelerometerOK ??
                target.AccelerometerOK;
            target.GyrometerOK = source.GyrometerOK ?? target.GyrometerOK;
            target.MagnetometerOK = source.MagnetometerOK ??
```

```

        target.MagnetometerOK;
    }
    catch (DocumentClientException ex)
    {
        if (ex.StatusCode == System.Net.HttpStatusCode.NotFound)
        {
            target = source;
        }
    }

    var collectionLink =
UriFactory.CreateDocumentCollectionUri(cosmosDBDatabase,
cosmosDBCollection);
    return await client.UpsertDocumentAsync(collectionLink, target);
}
}

```

This code uses the `IDocumentClient` interface to fetch a document from Azure Cosmos DB. If the document exists, the new state values are merged into the existing document. Otherwise, a new document is created. Both cases are handled by the `UpsertDocumentAsync` method.

This code is optimized for the case where the document already exists and can be merged. On the first telemetry message from a given drone, the `ReadDocumentAsync` method will throw an exception, because there is no document for that drone. After the first message, the document will be available.

Notice that this class uses dependency injection to inject the `IDocumentClient` for Azure Cosmos DB and an `IOptions<T>` with configuration settings. We'll see how to set up the dependency injection later.

ⓘ Note

Azure Functions supports an output binding for Azure Cosmos DB. This binding lets the function app write documents in Azure Cosmos DB without any code. However, the output binding won't work for this particular scenario, because of the custom upsert logic that's needed.

Error handling

As mentioned earlier, the `RawTelemetryFunction` function app processes a batch of messages in a loop. That means the function needs to handle any exceptions gracefully and continue processing the rest of the batch. Otherwise, messages might get dropped.

If an exception is encountered when processing a message, the function puts the message onto a dead-letter queue:

```
C#  
  
catch (Exception ex)  
{  
    logger.LogError(ex, "Error deserializing message",  
    message.SystemProperties.PartitionKey,  
    message.SystemProperties.SequenceNumber);  
    await deadLetterMessages.AddAsync(new DeadLetterMessage { Exception =  
        ex, EventData = message });  
}
```

The dead-letter queue is defined using an [output binding](#) to a storage queue:

```
C#  
  
[FunctionName("RawTelemetryFunction")]  
[StorageAccount("DeadLetterStorage")] // App setting that holds the  
connection string  
public async Task RunAsync(  
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection",  
    ConsumerGroup = "%EventHubConsumerGroup%")]EventData[] messages,  
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage>  
    deadLetterMessages, // output binding  
    ILogger logger)
```

Here the `Queue` attribute specifies the output binding, and the `StorageAccount` attribute specifies the name of an app setting that holds the connection string for the storage account.

Deployment tip: In the Resource Manager template that creates the storage account, you can automatically populate an app setting with the connection string. The trick is to use the [listKeys](#) function.

Here is the section of the template that creates the storage account for the queue:

JSON

```
{  
    "name": "  
[variables('droneTelemetryDeadLetterStorageQueueAccountName')]",  
    "type": "Microsoft.Storage/storageAccounts",  
    "location": "[resourceGroup().location]",  
    "apiVersion": "2017-10-01",  
    "sku": {
```

```
        "name": "[parameters('storageAccountType')]"  
    },
```

Here is the section of the template that creates the function app.

JSON

```
{  
    "apiVersion": "2015-08-01",  
    "type": "Microsoft.Web/sites",  
    "name": "[variables('droneTelemetryFunctionAppName')]",  
    "location": "[resourceGroup().location]",  
    "tags": {  
        "displayName": "Drone Telemetry Function App"  
    },  
    "kind": "functionapp",  
    "dependsOn": [  
        "[resourceId('Microsoft.Web/serverfarms',  
variables('hostingPlanName'))]",  
        ...  
    ],  
    "properties": {  
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms',  
variables('hostingPlanName'))]",  
        "siteConfig": {  
            "appSettings": [  
                {  
                    "name": "DeadLetterStorage",  
                    "value": "  
[concat('DefaultEndpointsProtocol=https;AccountName=',  
variables('droneTelemetryDeadLetterStorageQueueAccountName'),  
';AccountKey=',  
listKeys(variables('droneTelemetryDeadLetterStorageQueueAccountId'), '2015-  
05-01-preview').key1)]"  
                },  
                ...  
            ]  
        }  
    }  
}
```

This defines an app setting named `DeadLetterStorage` whose value is populated using the `listKeys` function. It's important to make the function app resource depend on the storage account resource (see the `dependsOn` element). This guarantees that the storage account is created first and the connection string is available.

Setting up dependency injection

The following code sets up dependency injection for the `RawTelemetryFunction` function:

C#

```
[assembly: FunctionsStartup(typeof(DroneTelemetryFunctionApp.Startup))]

namespace DroneTelemetryFunctionApp
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddOptions<StateChangeProcessorOptions>()
                .Configure< IConfiguration>((configSection, configuration) =>
            {
                configuration.Bind(configSection);
            });

            builder.Services.AddTransient<ITelemetrySerializer<DroneState>,
            TelemetrySerializer<DroneState>>();
            builder.Services.AddTransient<ITelemetryProcessor,
            TelemetryProcessor>();
            builder.Services.AddTransient<IStateChangeProcessor,
            StateChangeProcessor>();

            builder.Services.AddSingleton<IDocumentClient>(ctx => {
                var config = ctx.GetService< IConfiguration>();
                var cosmosDBEndpoint = config.GetValue< string >
                ("CosmosDBEndpoint");
                var cosmosDBKey = config.GetValue< string >("CosmosDBKey");
                return new DocumentClient(new Uri(cosmosDBEndpoint),
                cosmosDBKey);
            });
        }
    }
}
```

Azure Functions written for .NET can use the ASP.NET Core dependency injection framework. The basic idea is that you declare a startup method for your assembly. The method takes an `IFunctionsHostBuilder` interface, which is used to declare the dependencies for DI. You do this by calling `Add*` method on the `Services` object. When you add a dependency, you specify its lifetime:

- *Transient* objects are created each time they're requested.
- *Scoped* objects are created once per function execution.
- *Singleton* objects are reused across function executions, within the lifetime of the function host.

In this example, the `TelemetryProcessor` and `StateChangeProcessor` objects are declared as transient. This is appropriate for lightweight, stateless services. The `DocumentClient`

class, on the other hand, should be a singleton for best performance. For more information, see [Performance tips for Azure Cosmos DB and .NET](#).

If you refer back to the code for the [RawTelemetryFunction](#), you'll see there another dependency that doesn't appear in DI setup code, namely the `TelemetryClient` class that is used to log application metrics. The Functions runtime automatically registers this class into the DI container, so you don't need to register it explicitly.

For more information about DI in Azure Functions, see the following articles:

- [Use dependency injection in .NET Azure Functions](#)
- [Dependency injection in ASP.NET Core](#)

Passing configuration settings in DI

Sometimes an object must be initialized with some configuration values. Generally, these settings should come from app settings or (in the case of secrets) from Azure Key Vault.

There are two examples in this application. First, the `DocumentClient` class takes an Azure Cosmos DB service endpoint and key. For this object, the application registers a lambda that will be invoked by the DI container. This lambda uses the `IConfiguration` interface to read the configuration values:

C#

```
builder.Services.AddSingleton<IDocumentClient>(ctx => {
    var config = ctx.GetService< IConfiguration>();
    var cosmosDBEndpoint = config.GetValue< string >("CosmosDBEndpoint");
    var cosmosDBKey = config.GetValue< string >("CosmosDBKey");
    return new DocumentClient(new Uri(cosmosDBEndpoint), cosmosDBKey);
});
```

The second example is the `StateChangeProcessor` class. For this object, we use an approach called the [options pattern](#). Here's how it works:

1. Define a class `T` that contains your configuration settings. In this case, the Azure Cosmos DB database name and collection name.

C#

```
public class StateChangeProcessorOptions
{
    public string COSMOSDB_DATABASE_NAME { get; set; }
```

```
    public string COSMOSDB_DATABASE_COL { get; set; }  
}
```

2. Add the class `T` as an options class for DI.

```
C#  
  
builder.Services.AddOptions<StateChangeProcessorOptions>()  
    .Configure< IConfiguration>((configSection, configuration) =>  
    {  
        configuration.Bind(configSection);  
    });
```

3. In the constructor of the class that is being configured, include an `IOptions<T>` parameter.

```
C#  
  
public StateChangeProcessor(IDocumentClient client,  
    IOptions<StateChangeProcessorOptions> options)
```

The DI system will automatically populate the options class with configuration values and pass this to the constructor.

There are several advantages of this approach:

- Decouple the class from the source of the configuration values.
- Easily set up different configuration sources, such as environment variables or JSON configuration files.
- Simplify unit testing.
- Use a strongly typed options class, which is less error prone than just passing in scalar values.

GetStatus function

The other Functions app in this solution implements a simple REST API to get the last-known status of a drone. This function is defined in a class named `GetStatusFunction`. Here is the complete code for the function:

```
C#  
  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Azure.WebJobs;
```

```

using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;
using System.Security.Claims;
using System.Threading.Tasks;

namespace DroneStatusFunctionApp
{
    public static class GetStatusFunction
    {
        public const string GetDeviceStatusRoleName = "GetStatus";

        [FunctionName("GetStatusFunction")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Function, "get", Route =
null)]HttpRequest req,
            [CosmosDB(
                databaseName: "%COSMOSDB_DATABASE_NAME%",
                collectionName: "%COSMOSDB_DATABASE_COL%",
                ConnectionStringSetting = "COSMOSDB_CONNECTION_STRING",
                Id = "{Query.deviceId}",
                PartitionKey = "{Query.deviceId}")] dynamic deviceStatus,
            ClaimsPrincipal principal,
            ILogger log)
        {
            log.LogInformation("Processing GetStatus request.");

            if (!principal.IsAuthorizedByRoles(new[] {
                GetDeviceStatusRoleName }, log))
            {
                return new UnauthorizedResult();
            }

            string deviceId = req.Query["deviceId"];
            if (deviceId == null)
            {
                return new BadRequestObjectResult("Missing DeviceId");
            }

            if (deviceStatus == null)
            {
                return new NotFoundResult();
            }
            else
            {
                return new OkObjectResult(deviceStatus);
            }
        }
    }
}

```

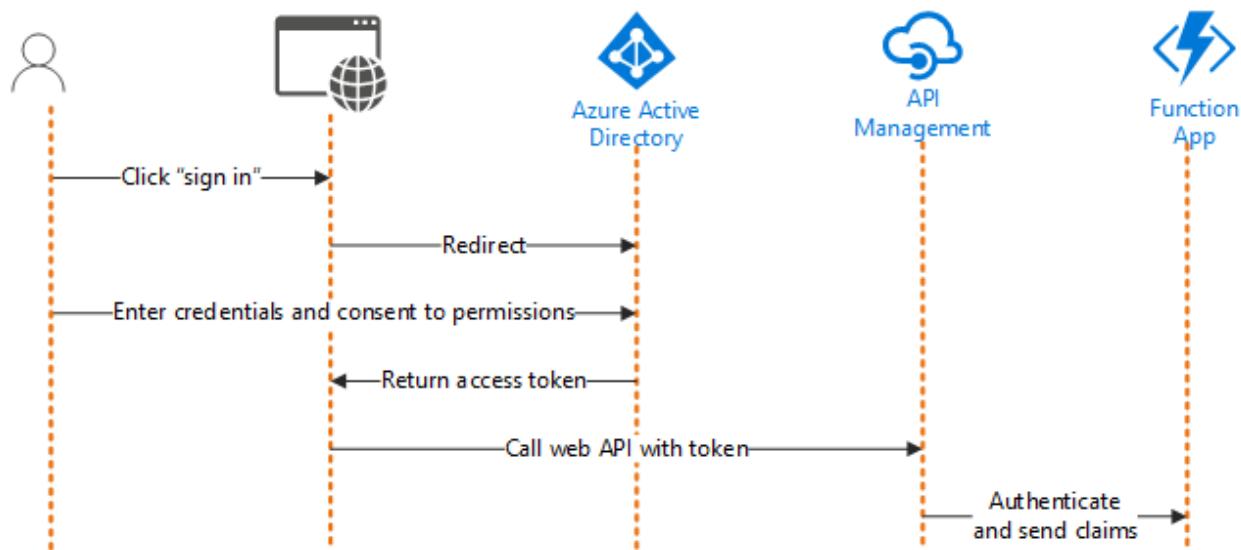
This function uses an HTTP trigger to process an HTTP GET request. The function uses an Azure Cosmos DB input binding to fetch the requested document. One consideration is that this binding will run before the authorization logic is performed inside the

function. If an unauthorized user requests a document, the function binding will still fetch the document. Then the authorization code will return a 401, so the user won't see the document. Whether this behavior is acceptable may depend on your requirements. For example, this approach might make it harder to audit data access for sensitive data.

Authentication and authorization

The web app uses Microsoft Entra ID to authenticate users. Because the app is a single-page application (SPA) running in the browser, the [implicit grant flow](#) is appropriate:

1. The web app redirects the user to the identity provider (in this case, Microsoft Entra ID).
2. The user enters their credentials.
3. The identity provider redirects back to the web app with an access token.
4. The web app sends a request to the web API and includes the access token in the Authorization header.



A Function application can be configured to authenticate users with zero code. For more information, see [Authentication and authorization in Azure App Service](#).

Authorization, on the other hand, generally requires some business logic. Microsoft Entra ID supports *claims based authentication*. In this model, a user's identity is represented as a set of claims that come from the identity provider. A claim can be any piece of information about the user, such as their name or email address.

The access token contains a subset of user claims. Among these are any application roles that the user is assigned to.

The `principal` parameter of the function is a [ClaimsPrincipal](#) object that contains the claims from the access token. Each claim is a key/value pair of claim type and claim

value. The application uses these to authorize the request.

The following extension method tests whether a `ClaimsPrincipal` object contains a set of roles. It returns `false` if any of the specified roles is missing. If this method returns `false`, the function returns HTTP 401 (Unauthorized).

C#

```
namespace DroneStatusFunctionApp
{
    public static class ClaimsPrincipalAuthorizationExtensions
    {
        public static bool IsAuthorizedByRoles(
            this ClaimsPrincipal principal,
            string[] roles,
            ILogger log)
        {
            var principalRoles = new HashSet<string>
(principal.Claims.Where(kvp => kvp.Type == "roles").Select(kvp =>
kvp.Value));
            var missingRoles = roles.Where(r =>
!principalRoles.Contains(r)).ToArray();
            if (missingRoles.Length > 0)
            {
                log.LogWarning("The principal does not have the required
{roles}", string.Join(", ", missingRoles));
                return false;
            }

            return true;
        }
    }
}
```

For more information about authentication and authorization in this application, see the [Security considerations](#) section of the reference architecture.

Next steps

Once you get a feel for how this reference solution works, learn best practices and recommendations for similar solutions.

- For a serverless event ingestion solution, see [Serverless event processing using Azure Functions](#).
- For a serverless web app, see [Serverless web application on Azure](#).

Azure Functions is just one Azure compute option. For help with choosing a compute technology, see [Choose an Azure compute service for your application](#).

Related resources

- For in-depth discussion on developing serverless solutions on premises as well as in the cloud, read [Serverless apps: Architecture, patterns, and Azure implementation](#).
- Read more about the [Event-driven architecture style](#).

CI/CD for a serverless application frontend on Azure

Azure Pipelines

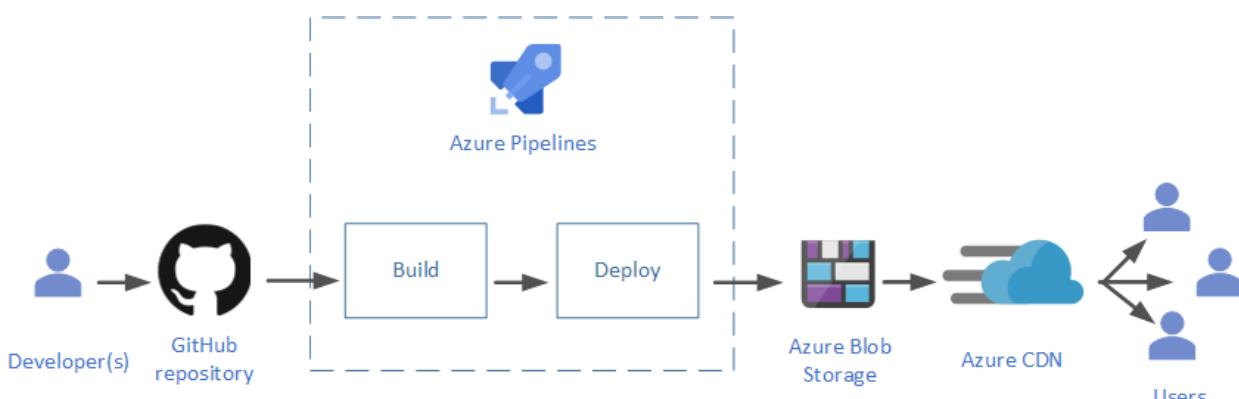
Serverless computing abstracts the servers, infrastructure, and operating systems, allowing developers to focus on application development. A robust *CI/CD* or *Continuous Integration/Continuous Delivery* of such applications allows companies to ship fully tested and integrated software versions within minutes of development. It provides a backbone of modern DevOps environment.

What does CI/CD actually stand for?

- Continuous Integration allows development teams to integrate code changes in a shared repository almost instantaneously. This ability, coupled with automated build and testing before the changes are integrated, ensures that only fully functional application code is available for deployment.
- Continuous Delivery allows changes in the source code, configuration, content, and other artifacts to be delivered to production, and ready to be deployed to end-users, as quickly and safely as possible. The process keeps the code in a *deployable state* at all times. A special case of this is *Continuous Deployment*, which includes actual deployment to end users.

This article discusses a CI/CD pipeline for the web frontend of a [serverless reference implementation](#). This pipeline is developed using Azure services. The web frontend demonstrates a modern web application, with client-side JavaScript, reusable server-side APIs, and pre-built Markup, alternatively called [Jamstack](#). You can find the code in [this GitHub repository](#). The readme describes the steps to download, build, and deploy the application.

The following diagram describes the CI/CD pipeline used in this sample frontend:



This article does not discuss the [backend deployment](#).

Prerequisites

To work with this sample application, make sure you have the following:

- A GitHub account.
- An Azure account. If you don't have one, you can try out a [free Azure account](#).
- An Azure DevOps organization. If you don't have one, you can try out a [basic plan](#), which includes DevOps services such as Azure Pipelines.

Use an online version control system

Version control systems keep track and control changes in your source code. Keeping your source code in an online version control system allows multiple development teams to collaborate. It is also easier to maintain than a traditional version control on premises. These online systems can be easily integrated with leading CI/CD services. You get the ability to create and maintain the source code in multiple directories, along with build and configuration files, in what is called a *repository*.

The project files for this sample application are kept in GitHub. If you don't have a GitHub account, read [this documentation to get started with GitHub repositories](#).

Automate your build and deploy

Using a CI/CD service such as [Azure Pipelines](#) can help you to automate the build and deploy processes. You can create multiple stages in the pipeline, each stage running based on the result of the previous one. The stages can run in either a [Windows or Linux container](#). The script must make sure the tools and environments are set properly in the container. Azure Pipelines can run a variety of build tools, and can work with quite a few [online version control systems](#).

Integrate build tools

Modern build tools can simplify your build process, and provide functionality such as pre-configuration, [minification](#) of the JavaScript files, and static site generation. Static site generators can build markup files before they are deployed to the hosting servers, resulting in a fast user experience. You can select from a variety of these tools, based on the type of your application's programming language and platform, as well as additional

functionality needed. [This article](#) provides a list of popular build tools for a modern application.

The sample is a React application, built using [Gatsby.js](#), which is a static site generator and front-end development framework. These tools can be run locally during development and testing phases, and then integrated with [Azure Pipelines](#) for the final deployment.

The sample uses the Gatsby file [gatsby-ssr.js](#) for rendering, and [gatsby-config.js](#) for site configuration. Gatsby converts all JavaScript files under the `pages` subdirectory of the `src` folder to HTML files. Additional components go in the `components` subdirectory. The sample also uses the [gatsby-plugin-typescript](#) plugin that allows using [TypeScript](#) for type safety, instead of JavaScript.

For more information about setting up a Gatsby project, see the official [Gatsby documentation](#).

Automate builds

Automating the build process reduces the human errors that can be introduced in manual processes. The file [azure-pipelines.yml](#) includes the script for a two-stage automation. [The Readme for this project](#) describes the steps required to set up the automation pipeline using Azure Pipelines. The following subsections show how the pipeline stages are configured.

Build stage

Since Azure Pipelines is [integrated with the GitHub repository](#), any changes in the tracked directory of the main branch trigger the first stage of the pipeline, the build stage:

YAML

```
trigger:
  batch: true
  branches:
    include:
    - main
  paths:
    include:
    - src/ClientApp
```

The following snippet illustrates the start of the build stage, which starts an Ubuntu container to run this stage.

YAML

```
stages:
- stage: Build
  jobs:
  - job: WebsiteBuild
    displayName: Build Fabrikam Drone Status app
    pool:
      vmImage: 'Ubuntu-16.04'
      continueOnError: false
  steps:
```

This is followed by *tasks* and *scripts* required to successfully build the project. These include the following:

- Installing Node.js and setting up environment variables,
- Installing and running Gatsby.js that builds the static website:

YAML

```
- script: |
  cd src/ClientApp
  npm install
  npx gatsby build
  displayName: 'gatsby build'
```

- installing and running a compression tool named *brotli*, to [compress the built files](#) before deployment:

YAML

```
- script: |
  cd src/ClientApp/public
  sudo apt-get install brotli --install-suggests --no-install-recommends -q --assume-yes
  for f in $(find . -type f \(\ -iname '*.html' -o -iname '*.map' -o -iname '*.js' -o -iname '*.json' \)); do brotli $f -Z -j -f -v && mv ${f}.br $f; done
  displayName: 'enable compression at origin level'
```

- Computing the version of the current build for [cache management](#),
- Publishing the built files for use by the [deploy stage](#):

YAML

```
- task: PublishPipelineArtifact@1
  inputs:
    targetPath: 'src/ClientApp/public'
    artifactName: 'drop'
```

A successful completion of the build stage tears down the Ubuntu environment, and triggers the deploy stage in the pipeline.

Deploy stage

The deploy stage runs in a new Ubuntu container:

YAML

```
- stage: Deploy
  jobs:
    - deployment: WebsiteDeploy
      displayName: Deploy Fabrikam Drone Status app
      pool:
        vmImage: 'Ubuntu-16.04'
      environment: 'fabrikamdronestatus-prod'
      strategy:
        runOnce:
          deploy:
            steps:
```

This stage includes various deployment tasks and scripts to:

- Download the build artifacts to the container (which happens automatically as a consequence of using `PublishPipelineArtifact` in the build stage),
- Record the build release version, and update in the GitHub repository,
- Upload the website files to Blob Storage, in a new folder corresponding to the new version, and
- Change the CDN to point to this new folder.

The last two steps together replicate a cache purge, since older folders are no longer accessible by the CDN edge servers. The following snippet shows how this is achieved:

YAML

```
- script: |
  az login --service-principal -u $(azureArmClientId) -p
  $(azureArmClientSecret) --tenant $(azureArmTenantId)
    # upload content to container versioned folder
    az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" -
  -destination "\$web\$(releaseSemVer)" --account-name
```

```

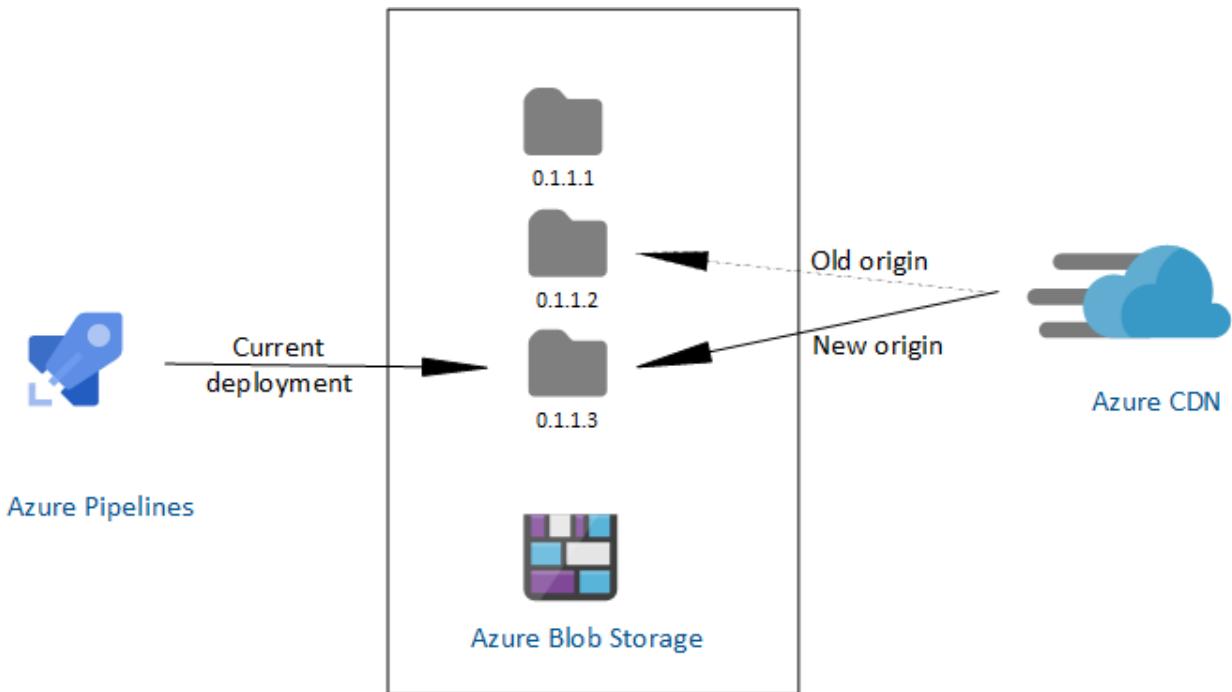
$(azureStorageAccountName) --content-encoding br --pattern "*.html" --
content-type "text/html"
    az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --
-destination "\$web\$(releaseSemVer)" --account-name
$(azureStorageAccountName) --content-encoding br --pattern "*.js" --content-
type "application/javascript"
    az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --
-destination "\$web\$(releaseSemVer)" --account-name
$(azureStorageAccountName) --content-encoding br --pattern "*.js.map" --
content-type "application/octet-stream"
    az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --
-destination "\$web\$(releaseSemVer)" --account-name
$(azureStorageAccountName) --content-encoding br --pattern "*.json" --
content-type "application/json"
    az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --
-destination "\$web\$(releaseSemVer)" --account-name
$(azureStorageAccountName) --pattern "*.txt" --content-type "text/plain"
    # target new version
    az cdn endpoint update --resource-group $(azureResourceGroup)
--profile-name $(azureCdnName) --name $(azureCdnName) --origin-path
'$(releaseSemVer)'
    AZURE_CDN_ENDPOINT_HOSTNAME=$(az cdn endpoint show --resource-
group $(azureResourceGroup) --name $(azureCdnName) --profile-name
$(azureCdnName) --query hostName -o tsv)
    echo "Azure CDN endpoint host ${AZURE_CDN_ENDPOINT_HOSTNAME}"
    echo '##vso[task.setvariable
variable=azureCndEndpointHost]'$AZURE_CDN_ENDPOINT_HOSTNAME
        displayName: 'upload to Azure Storage static website hosting and
purge Azure CDN endpoint'

```

Atomic deploys

Atomic deployment ensures that the users of your website or application always get the content corresponding to the same version.

In the sample CI/CD pipeline, the website contents are deployed to the Blob storage, which acts as [the origin server for the Azure CDN](#). If the files are updated in the same *root folder* in the blob, the website will be served inconsistently. Uploading to a new versioned folder as shown in the preceding section solves this problem. The users either get *all or nothing* of the new successful build, since the CDN points to the new folder as the origin, only after all files are successfully updated.



The advantages of this approach are as follows:

- Since new content is not available to users until the CDN points to the new origin folder, it results in an atomic deployment.
- You can easily roll back to an older version of the website if necessary.
- Since the origin can host multiple versions of the website side by side, you can fine-tune the deployment by using techniques such as allowing preview to certain users before wider availability.

Host and distribute using the cloud

A content delivery network (CDN) is a set of distributed servers that speed up the content delivery to users over a vast geographical area. Every user gets the content from the server nearest to them. The CDN accesses this content from an *origin* server, and caches it to *edge* servers at strategic locations. The sample CI/CD in this article uses [Azure CDN](#), pointing to website content hosted on [Azure Blob Storage](#) as the origin server. The Blob storage is [configured for static website hosting](#). For a quick guide on how to use Azure CDN with Azure Blob Storage, read [Integrate an Azure storage account with Azure CDN](#).

The following are some strategies that can improve the CDN performance.

Compress the content

Compressing the files before serving improves file transfer speed and increases page-load performance for the end users.

There are two ways to do this:

1. **On the fly in the CDN edge servers.** This improves bandwidth consumption by a significant percentage, making the website considerably faster than without compression. It is relatively easy to [enable this type of compression on Azure CDN](#). Since it's controlled by the CDN, you cannot choose or configure the compression tool. Use this compression if website performance is not a critical concern.
2. **Pre-compressing before reaching the CDN**, either on your origin server, or before reaching the origin. Pre-compressing further improves the run time performance of your website, since it's done before being fetched by the CDN. The sample CI/CD pipeline compresses the built files in the deployment stage, using [brotli](#), as shown in the [build section above](#). The advantages of using pre-compression are as follows:
 - a. You can use any compression tool that you're comfortable with, and further fine-tune the compression. CDNs may have limitations on the tools they use.
 - b. Some CDNs limit file sizes that can be compressed on the fly, causing a performance hit for larger files. Pre-compression sets no limits on the file size.
 - c. Pre-compressing in the deployment pipeline results in less storage required at the origin.
 - d. This is faster and more efficient than the CDN compression.

For more information, see [Improve performance by compressing files in Azure CDN](#).

Dynamic site acceleration

Using CDN is optimal for static content, which can be safely cached at the edge servers. However, dynamic web sites require the server to generate content based on user response. This type of content cannot be cached on the edge, requiring a more involved end-to-end solution that can speed up the content delivery. [Dynamic site acceleration by Azure CDN](#) is one such solution that measurably improves the performance of dynamic web pages.

This sample enables dynamic site acceleration as shown in [this section of the readme](#).

Manage website cache

CDNs use caching to improve their performance. Configuring and managing this cache becomes an integral part of your deployment pipeline. The Azure CDN documentation shows several ways to do this. You can set caching rules on your CDN, as well as [configure the time-to-live for the content in the origin server](#). Static web content can be cached for a long duration, since it may not change too much over time. This reduces

the overhead of accessing the single origin server for every user request. For more information, see [How caching works](#).

Cache purge

The CDN caches the website files at the edge servers until their time-to-live expires. These are updated for a new client request, only after their time-to-live has expired. Purging your CDN cache is required to guarantee that every user gets the latest live website files, especially if the deployment happens at the same CDN origin folder. Azure CDN allows you to [purge the cache from the Azure portal](#).

A better approach is to invalidate this cache by [using versioning during deployment](#). An explicit cache purge or expiration usually causes the CDN to retrieve newer versions of the web content. However, since the CDN always points to the latest version of the deployment, it improves caching in the following manner:

1. The CDN validates the index.html against the origin, for every new website instance.
2. Except for the index.html and 404.html, all other website files are fingerprinted and cached for a year. This is based on the assumption that resources such as images and videos, do not need frequent changes. If these files are updated and rebuilt, their names are updated by a new fingerprint GUID. This results in an update to the index.html with an updated reference to the changed resource file. The CDN then retrieves the updated index.html, and since it does not find the reference resource file in its cache, it also retrieves the changed resource files.

This ensures that the CDN always gets new updated files, and removes the need to purge the cache for a new build.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Dhanashri Kshirsagar](#) | Senior Content PM

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- Now that you understand the basics, follow [this readme](#) to set up and execute the CI/CD pipeline.
- Learn best practices for using content delivery networks (CDNs)

Monitor a distributed system by using Application Insights and OpenCensus

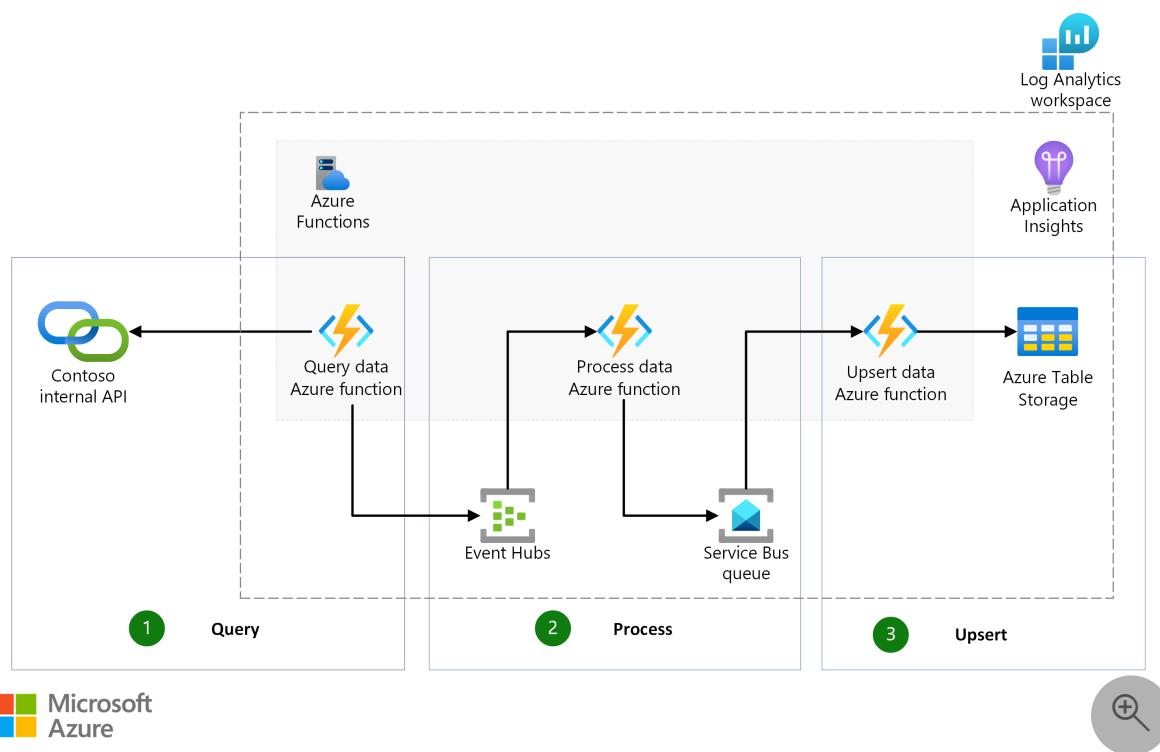
Azure Event Hubs Azure Functions Azure Service Bus Azure Monitor

This article describes a distributed system that's created with Azure Functions, Azure Event Hubs, and Azure Service Bus. It provides details about how to monitor the end-to-end system by using [OpenCensus for Python](#) and Application Insights. This article also introduces distributed tracing and explains how it works by using Python code examples. The fictional company, Contoso, is used in the architecture to help describe the scenario.

ⓘ Note

OpenCensus and OpenTelemetry are merging, but OpenCensus is still the recommended tool to monitor Azure Functions. OpenTelemetry for Azure is in preview and **some features aren't available yet**.

Architecture



Download a [Visio file](#) of this architecture.

Workflow

1. **Query.** A timer-triggered Azure function queries the Contoso internal API to get the latest sales data once a day. The function uses the [Azure Event Hubs output binding](#) to send the unstructured data as events.
2. **Process.** Event Hubs triggers an Azure function that processes and formats the unstructured data to a pre-defined structure. The function publishes one message to Service Bus per asset that needs to be imported via the [Service Bus output binding](#).
3. **Upsert.** Service Bus triggers an Azure function that consumes messages from the queue and runs an upsert operation in the common company storage.

It's important to consider potential operation failures of this architecture. Some examples include:

- The internal API is unavailable, which leads to an exception that's raised by the query data Azure function in step one of the architecture.
- In step two, the process data Azure function encounters data that's outside of the conditions or parameters of the function.
- In step three, the upsert data Azure function fails. After several retries, the messages from the Service Bus queue go in the [dead-letter queue](#), which is a secondary queue that holds messages that can't be processed or delivered to a receiver after a predefined number of retries. Then the messages can follow an established automatic process, or they can be handled manually.

Components

- [Azure Functions](#) is a serverless service that manages your applications.
- [Application Insights](#) is a feature of Azure Monitor that monitors applications in development, test, and production. Application Insights analyzes how an application performs, and it reviews application run data to determine the cause of an incident.
- [Azure Table Storage](#) is a service that stores nonrelational structured data (structured NoSQL data) in the cloud and provides a key/attribute store with a schemaless design.
- [Event Hubs](#) is a scalable event ingestion service that can receive and process millions of events per second.

- [OpenCensus](#) is a set of open-source libraries that you can use to collect distributed traces, metrics, and logging telemetry. This architecture uses the Python implementation of OpenCensus.
- [Service Bus](#) is a fully managed message broker with message queues and publish-subscribe topics.

Scenario details

Distributed systems are made of loosely coupled components. It can be difficult to understand how the components communicate and to fully perceive the end-to-end journey of a user request. This architecture helps you see how components are connected.

Like many companies, Contoso needs to ingest on-premises or third-party data in the cloud while also collecting data about their sales by using services and in-house tools. In this architecture, a department at Contoso built an internal API that exposes the unstructured data, and they ingest the data into common storage. The common storage contains structured data from every department. The architecture shows how Contoso extracts, processes, and ingests that metadata in the cloud.

When you build a system, especially a distributed system, it's important to make it observable. An observable system:

- Provides a holistic view of the health of the distributed application.
- Measures the operational performance of the system.
- Identifies and diagnoses failures so you can quickly resolve an issue.

Distributed tracing

In this architecture, the system is a chain of microservices. Each microservice can fail independently for various reasons. When that happens, it's important to understand what happened so you can troubleshoot. It's helpful to isolate an end-to-end transaction and follow the journey through the app stack, which consists of services or microservices. This method is called *distributed tracing*.

The following sections describe how to set up distributed tracing in the architecture. Select the following **Deploy to Azure** button to deploy the infrastructure and the Azure function app.

 **Note**

There isn't an internal API in the architecture, so a read of an Azure file replaces the call to an API.



Deploy to Azure



Traces and spans

A transaction is represented by a *trace*, which is a collection of [spans](#). For example, when you select the purchase button to place an order on an e-commerce website, several subsequent operations take place. Some possible operations include:

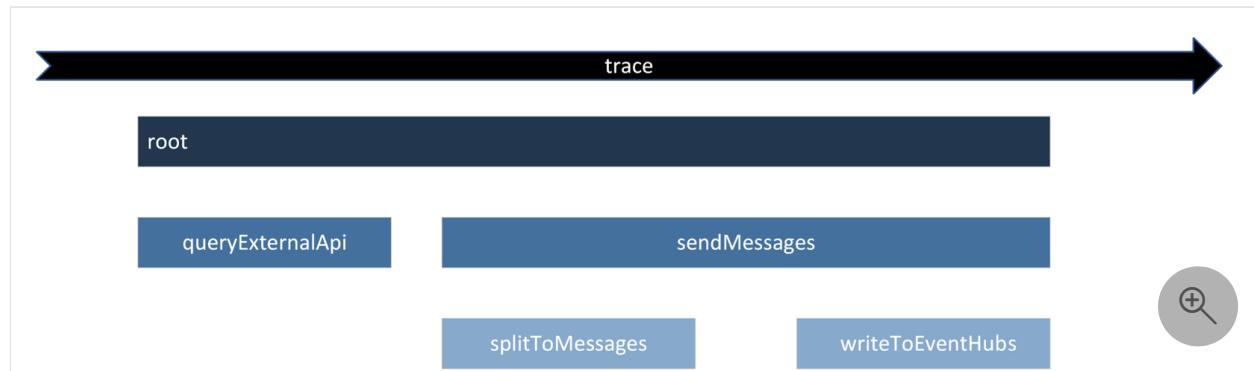
- A POST request submits to the API, which then redirects you to a “waiting page.”
- Writing logs with contextual information.
- An external call to web-based software to request a billing page.

Each of these operations can be part of a span. The trace is a complete description of what happens when you select the purchase button.

Similarly, in this architecture, when the query data Azure function triggers to start the daily ingestion of the sales data, a trace is created that contains multiple spans:

- A span to confirm the trigger details.
- A span to query the internal API.
- A span to create and send an event to Event Hubs.

A span can have child spans. For example, the following image shows the query data Azure function as a trace:



- The sendMessages span is split into two child spans: splitToMessages and writeToEventHubs. The sendMessages span requires those two suboperations to send messages.
- All spans are children of a root span.

- Spans give you an easy way to describe all parts involved in the query step of the query data Azure function. Each Azure function is a trace. So an end-to-end pass through Contoso's ingestion system is the union of three traces, which are the three Azure functions. When you combine the three traces and their telemetry, you build the end-to-end journey and describe all parts of the architecture.

Tracers and the W3C trace context

A *tracer* is an object that holds contextual information. Ideally, that contextual information propagates as data transits through the Azure functions. To propagate the information, the OpenCensus extension uses [the W3C trace context](#).

As its documentation states, the W3C trace context is a "specification that defines standard HTTP headers and a value format to propagate context information that enables distributed tracing scenarios."

A component of the system, such as a function, can create a tracer with the context of the previous component that's making the call by reading the trace parent. The format of a trace is:

Traceparent: [version]-[traceId]-[parentId]-[traceFlags]

For instance, if traceparent = 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-00

base16(version) = 00

base16(traceId) = 0af7651916cd43dd8448eb211c80319c

base16(parentId) = b7ad6b7169203331

base16(traceFlags) = 00

The trace ID and parent ID are the most important fields. The trace ID is a globally unique identifier of a trace. The parent ID is a globally unique identifier of a span. That span is part of the trace that the trace ID identifies.

For more information, see [Traceparent header](#).

In the remaining sections of this article, it's assumed that the base16(version) and base16(traceFlags) are set to 00.

Create a tracer with the OpenCensus extension

Use the OpenCensus extension that's specific to Azure Functions. Don't use the OpenCensus package that you might use in other cases (for example, [Python Webapps](#)).

Azure Functions offers many input and output bindings, and each binding has a different way of embedding the trace parent. For this architecture, when events and messages are consumed, two Azure functions are triggered.

Before the two functions can trigger:

1. The context (characterized by the identifier of the trace and the identifier of the current span) must be embedded in a trace parent in the W3C trace context format. This embedding is dependent on the nature of the output binding. For instance, the architecture uses Event Hubs as a messaging system. The trace parent is encoded into bytes and embedded in the sent event as the diagnostic ID property, which achieves the right trace context in the output binding.

Two spans can be linked even if they're not parent and child. For distributed tracing, the current span points to the next one. Creating a [link](#) establishes this relationship.

The [Azure Functions Worker](#) package manages the embedding and linking for you.

2. An Azure function in the middle of the end-to-end flow extracts the contextual information from the passed-on trace parent. Use the OpenCensus extension for Azure Functions for this step. Instead of adding this process in the code of each Azure function, the OpenCensus extension implements a preinvocation hook on the function app level.

The preinvocation hook:

- Creates a span context object that holds the information of the previous span and triggers the Azure function. See a visual example of this step [in the next section](#).
- Creates a tracer that contains the span context and creates a new trace for the triggered Azure function.
- Injects the tracer in the [Azure function execution context](#).

To ensure the traces appear in Application Insights, you must call the `configure` method to create and configure an [Azure exporter](#), which exports telemetry.

The extension is at the app level, so the steps in this section apply to all Azure functions in a function app.

Understand and structure the code

In this architecture, the code in the Azure functions is structured with spans. In Python, create an OpenCensus span by using the `with` statement to access the span context part of the tracer that's injected in the Azure function execution context. The following string provides the details of the current span and its parents:

Python

```
with contexttracer.span("nameSpan"):  
    # DO SOMETHING WITHIN THAT SPAN
```

The following code shows details of the query data Azure function:

Python

```
import datetime  
import logging  
  
import azure.functions as func  
from opencensus.extension.azure.functions import OpenCensusExtension  
from opencensus.trace import config_integration  
  
OpenCensusExtension.configure()  
config_integration.trace_integrations(['requests'])  
config_integration.trace_integrations(['logging'])  
  
def main(timer: func.TimerRequest, outputEventHubMessage: func.Out[str],  
context: func.Context) -> None:  
  
    utc_timestamp = datetime.datetime.utcnow().replace(  
        tzinfo=datetime.timezone.utc).isoformat()  
  
    if timer.past_due:  
        logging.info('The timer is past due!')  
  
    logging.info(f"Query Data Azure Function triggered. Current tracecontext  
is: {context.trace_context.Traceparent}")  
    with contexttracer.span("queryExternalCatalog"):  
        logging.info('querying the external catalog')  
        content = {"key_content_1": "thisisavalue1"}  
        content = json.dumps(content)  
  
        with contexttracer.span("sendMessage"):  
            logging.info('reading the external catalog')  
  
            with contexttracer.span("splitToMessages"):  
                # Do sthg  
                logging.info('splitting to messages')  
  
            with contexttracer.span("setMessages"):
```

```
logging.info('sending messages')
outputEventHubMessage.set(content)

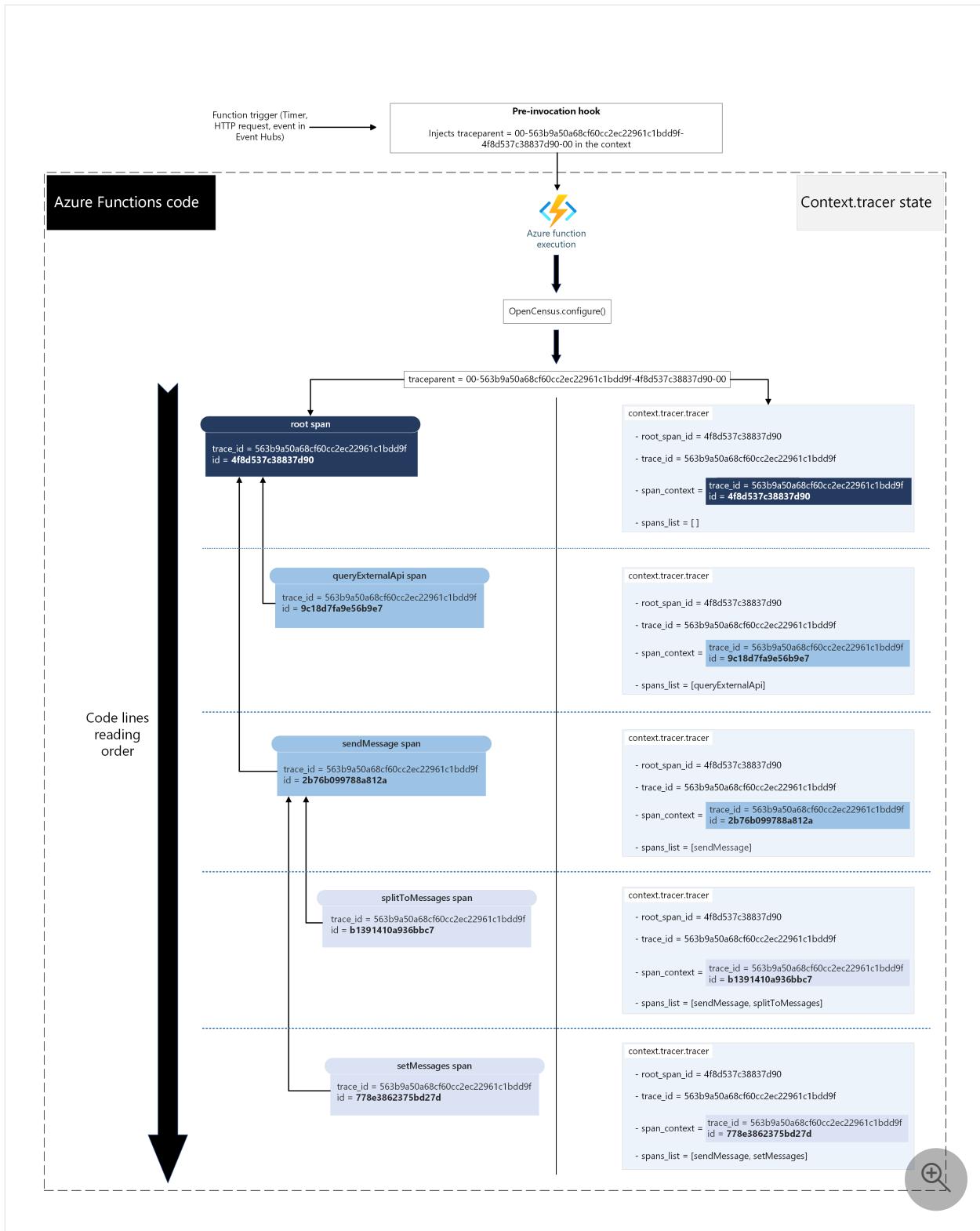
logging.info('Python timer trigger function ran at %s', utc_timestamp)
```

The main points in this code are:

- An `OpenCensusExtension.configure` call. Perform this call in only one Azure function per function app. This action configures the Azure exporter to export Python telemetry, such as logs, metrics, and traces, to Application Insights.
- The OpenCensus `requests` and `logging` [integrations](#) to configure the telemetry collection from the request and logging modules for HTTP calls.
- There are five spans:
 - A root span that's part of the tracer that's injected in the context before the execution
 - `queryExternalCatalog`
 - `sendMessage`
 - `splitToMessages` (a child of `sendMessage`)
 - `setMessages` (a child of `sendMessage`)

Tracers and spans

The following diagram shows how every time a span is created, the span context of the tracer is updated.



In the previous diagram:

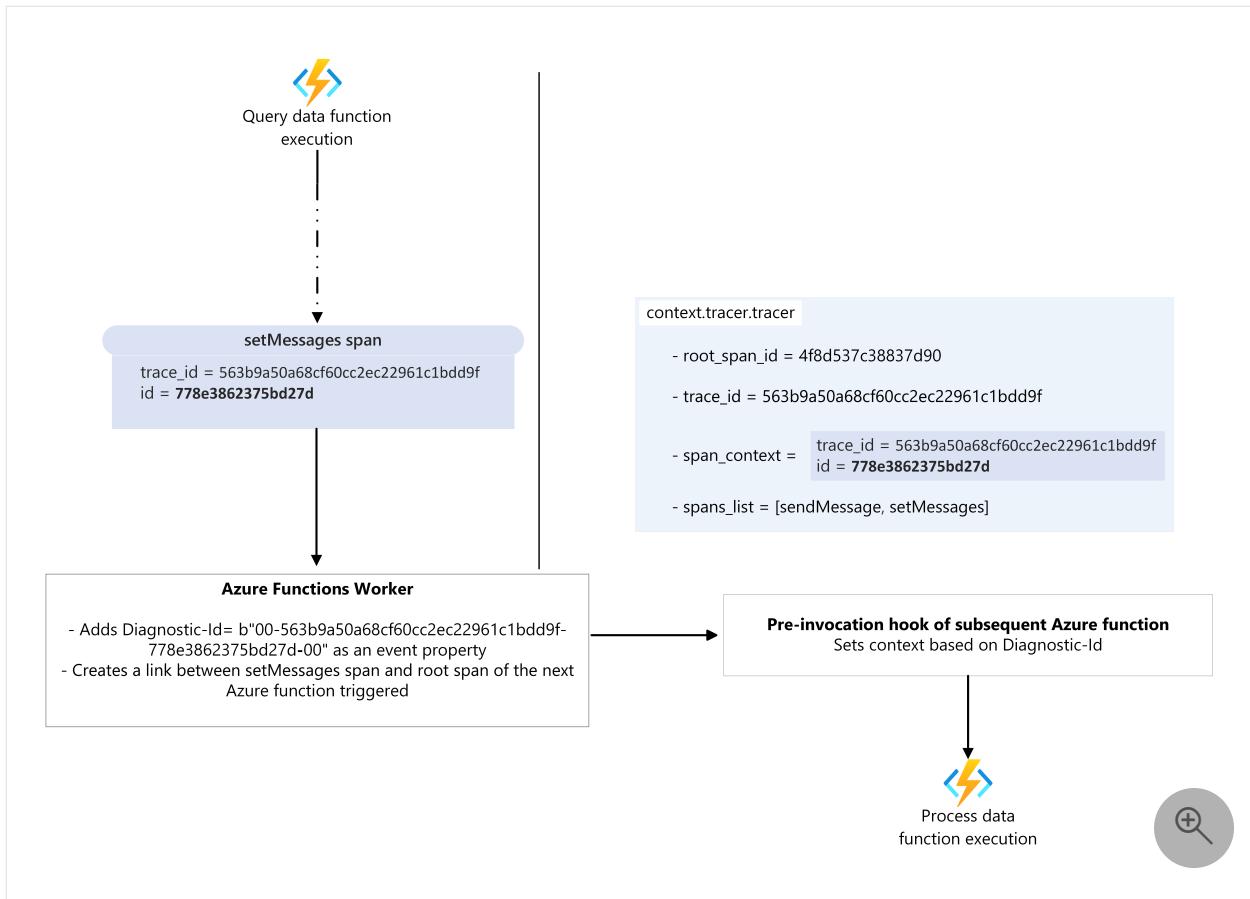
1. **An Azure function is triggered.** A trace parent is injected in the tracer context object with a preinvocation hook, which is called by the Python worker before the function runs.
2. **An Azure function is run.** The `OpenCensusExtension.configure` method is called, which initializes an Azure exporter and enables trace writing to Application Insights.

The following details explain the relationship between a tracer and a span in this architecture:

- The tracer object of the Azure function context contains a **span_context** field that describes the root span.
- Every time you create a span in code, it creates a new globally unique identifier and updates the **span_context** property in the tracer object of the execution context.
- The **span_context** field contains the **trace_id** and **id** fields.
- The **trace_id** never gets updated, but the **id** updates to the generated unique identifier.
- In the previous diagram, the root span has two child spans: `queryExternalApi` and `sendMessage`.
 - The `queryExternalApi` span and `sendMessage` span have a new span ID that's different from the **root_span_id**.
 - The `sendMessage` span has two child spans: `splitToMessages` and `setMessages`. Their span IDs update in the **span_context** field of the tracer object of the context.
- To capture the relationship between a child span and its parent, the **spans_list** field provides the lineage of spans in list form. In the `splitToMessages` span, the **spans_list** field contains `sendMessage` (the parent span) and `splitToMessages` (the current span). This parent/child relationship is how you create the chain of isolated operations within the execution of an Azure function.

Chain the functions by using the context field

Now that the chain of operations is organized in one Azure function, you can chain it to the subsequent operations performed by the next Azure function.



In the previous diagram:

- The setMessages span is the last span of the query data Azure function. The code within the span sends a message to Event Hubs and triggers the subsequent Azure function. The **span_context** field of the context tracer object contains the information related to this span. That information is tied to the query data Azure function's context.
- Azure Functions Worker adds a bytes-encoded **Diagnostic-Id** in the properties of the sent event and creates a [link](#) to the root span of the subsequent Azure function.
- The preinvocation hook of the subsequent process data Azure function reads the **Diagnostic-Id** and sets the context, which chains the Azure functions, and they're executed separately.

When the process data Azure function sends a message to the Service Bus queue, context is passed in the same way.

When the monitoring configurations are in place, use the Application Insights features to query and visualize the end-to-end transactions.

Types of telemetry

There are several types of telemetry available in Application Insights. The code in this architecture generates the following telemetry:

- **Request** telemetry emits when you call an HTTP or trigger an Azure function. The entry to Contoso's system has a timer trigger for the query data Azure function that emits request telemetry.
- **Dependency** telemetry emits when you make a call to an Azure service or an external service. When the Azure function writes an event to Event Hubs, it emits dependency telemetry.
- **Trace** telemetry emits from logs generated by Azure Functions runtime and Azure Functions. The logging inside the Azure function emits trace telemetry.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Raouf Aliouat](#) | Software Engineer II

Other contributors:

- [Julien Corioland](#) | Principal Software Engineer
- [Benjamin Guinebertière](#) | Principal Software Engineering Manager
- [Jodi Martis](#) | Technical Writer
- [Adina Stoll](#) | Software Engineer II

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- An example of a system that uses the presented approach: [Synchronization framework for metadata ingestion from external catalogs in Microsoft Purview](#)
- [Azure Monitor overview](#)
- [Observability in microservices](#)
- [Distributed tracing and telemetry correlation](#)
- [Understand operation IDs and operation links in Event Hubs](#)
- [OpenCensus Azure Monitor exporters](#)
- [Metadata ingestion from external catalogs in Microsoft Purview](#)

Related resources

- [Integrate Event Hubs with serverless functions on Azure](#)
- [Monitor Azure Functions and Event Hubs](#)
- [Monitor serverless event processing](#)
- [Performance and scale guidance for Event Hubs and Azure Functions](#)

Monitor serverless event processing

Article • 10/28/2022

This article provides guidance on monitoring [serverless](#) event-driven architectures.

Monitoring provides insight into the behavior and health of your systems. It helps you build a holistic view of the environment, retrieve historic trends, correlate diverse factors, and measure changes in performance, consumption, or error rate. You can use monitoring to define alerts when conditions occur that could impact the quality of your service, or when conditions of particular interest to your specific environment arise.

This article demonstrates using [Azure Monitor](#) to monitor a serverless application built using [Event Hubs](#) and [Azure Functions](#). It discusses useful metrics to monitor, describes how to integrate with [Application Insights](#) and capture custom metrics, and provides code samples.

Assumptions

This article assumes you have an architecture like the one described in the [Serverless event processing reference architecture](#). Basically:

- Events arrive at Azure Event Hubs.
- A Function App is triggered to handle the event.
- Azure Monitor is available for use with your architecture.

Metrics from Azure Monitor

First we need to decide which metrics will be needed before we can begin to formulate useful insights about the architecture. Each resource performs different tasks, and in turn generates different metrics.

These metrics from Event Hub will be of interest to capture useful insights:

- Incoming requests
- Outgoing requests
- Throttled requests
- Successful requests
- Incoming messages
- Outgoing messages
- Captured messages
- Incoming bytes

- Outgoing bytes
- Captured bytes
- User errors

Similarly, these metrics from Azure Functions will be of interest to capture useful insights:

- Function execution count
- Connections
- Data in
- Data out
- HTTP server errors
- Requests
- Requests in application queue
- Response time

Using diagnostics logging to capture insights

When analyzed together, the above metrics can be used to formulate and capture the following insights:

- Rate of requests processed by Event Hubs
- Rate of requests processed by Azure Functions
- Total Event Hub throughput
- User errors
- Duration of Azure Functions
- End-to-end latency
- Latency at each stage
- Number of messages lost
- Number of messages processed more than once

To ensure that Event Hubs captures the necessary metrics, we must first enable diagnostic logs (which are disabled by default). We must then select which logs are desired and configure the correct Log Analytics workspace as the destination for them.

The log and metric categories that we are interested in are:

- OperationalLogs
- AutoScaleLogs
- KafkaCoordinatorLogs *(for Apache Kafka workloads)*
- KafkaUserErrorLogs *(for Apache Kafka workloads)*
- EventHubVNetConnectionEvent

- AllMetrics

Azure documentation provides instructions on how to [Set up diagnostic logs for an Azure event hub](#). The following screenshot shows an example *Diagnostic setting* configuration panel with the correct log and metric categories selected, and a Log Analytics workspace set as the destination. (If an external system is being used to analyze the logs, the option to *Stream to an event hub* can be used instead.)

Diagnostic setting Save Discard Delete Provide feedback

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. [Learn more about the different log categories and contents of those logs](#)

Diagnostic setting name: eh-log-analytics

Category details

log

- ArchiveLogs
- OperationalLogs
- AutoScaleLogs
- KafkaCoordinatorLogs
- KafkaUserErrorLogs
- EventHubVNetConnectionEvent
- CustomerManagedKeyUserLogs

metric

- AllMetrics

Destination details

Send to Log Analytics workspace

Subscription: [REDACTED]

Log Analytics workspace: samltbkkend-loganalytics (southcentralus)

Archive to a storage account

Stream to an event hub

ⓘ Note

In order to utilize log diagnostics to capture insights, you should create event hubs in different namespaces. This is because of a constraint in Azure.

The Event Hubs set in a given Event Hubs namespace is represented in Azure Monitor metrics under a dimension called `EntityName`. In the Azure portal, data for a specific event hub normally can be viewed on that instance of Azure Monitor. But when the metrics data is routed to the log diagnostics, there is currently no way to view data per event hub by filtering on the `EntityName` dimension.

As a workaround, creating event hubs in different namespaces helps make it possible to locate metrics for a specific hub.

Using Application Insights

You can enable Application Insights to capture metrics and custom telemetry from Azure Functions. This allows you to define analytics that suit your own purposes, providing another way to get important insights for the serverless event processing scenario.

This screenshot shows an example listing of custom metrics and telemetry within Application Insights:

timestamp [UTC]	message	severityLevel	itemType	customDimensions
11/10/2020, 2:43:28.754 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Re...	1	trace	{"prop_(OriginalFormat)": "Exec...
11/10/2020, 2:43:28.754 AM	Trigger Details: PartitionId: 14, Offset: 25800738240, EnqueueTimeUtc...	1	trace	{"prop_(OriginalFormat)": "Trigg...
11/10/2020, 2:43:28.754 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Re...	1	trace	{"prop_(OriginalFormat)": "Exec...
11/10/2020, 2:43:28.754 AM	Trigger Details: PartitionId: 14, Offset: 25800738240, EnqueueTimeUtc...	1	trace	{"prop_(OriginalFormat)": "Trigg...
11/10/2020, 2:43:28.753 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Re...	1	trace	{"prop_(OriginalFormat)": "Exec...
11/10/2020, 2:43:28.716 AM	Executed 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succ...	1	trace	{"prop_(OriginalFormat)": "Exec...
11/10/2020, 2:43:28.716 AM	Read Payload from message: {"Body": [123, 34, 107, 101, 121, 34, 58, 34, 97, ...}	1	trace	{"prop_(OriginalFormat)": "Read...
11/10/2020, 2:43:28.715 AM	Executed 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succ...	1	trace	{"prop_(OriginalFormat)": "Exec...
11/10/2020, 2:43:28.715 AM	Read Payload from message: {"Body": [123, 34, 107, 101, 121, 34, 58, 34, 53, ...}	1	trace	{"prop_(OriginalFormat)": "Read...

Default custom metrics

In Application Insights, custom metrics for Azure Functions are stored in the `customMetrics` table. It includes the following values, spanned over a timeline for different function instances:

- `AvgDurationMs`
- `MaxDurationMs`
- `MinDurationMs`
- `Successes`
- `Failures`
- `SuccessRate`
- `Count`

These metrics can be used to efficiently calculate the aggregated averages across the multiple function instances that are invoked in a run.

This screenshot shows what these default custom metrics look like when viewed in Application Insights:

timestamp [UTC]	name	value	valueCount	valueSum	valueMin
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Count	1,000	1	1,000	1,000
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent AvgDurationMs	84.757	1	84.757	84.757
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent MaxDurationMs	276.234	1	276.234	276.234
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent MinDurationMs	11.744	1	11.744	11.744
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Successes	1,000	1	1,000	1,000
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Failures	0	1	0	0
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent SuccessRate	100	1	100	100

Custom messages

Custom messages logged in the Azure Function code (using the `ILogger`) are obtained from the Application Insights `traces` table.

The `traces` table has the following important properties (among others):

- timestamp
 - cloud_RoleInstance
 - operation_Id
 - operation_Name
 - message

Here is an example of what a custom message might look like in the Application Insights interface:

If the incoming Event Hub message or `EventData[]` is logged as a part of this custom `ILogger` message, then that is also made available in Application Insights. This can be very useful.

For our serverless event processing scenario, we log the JSON serialized message body that's received from the event hub. This allows us to capture the raw byte array, along with `SystemProperties` like `x-opt-sequence-number`, `x-opt-offset`, and `x-opt-enqueued-time`. To determine when each message was received by the Event Hub, the `x-opt-enqueued-time` property is used.

Sample query:

Kusto

```
traces
| where timestamp between(min_t .. max_t)
| where message contains "Body"
| extend m = parse_json(message)
| project timestamp = todatetime(m.SystemProperties["x-opt-enqueued-time"])
```

The sample query would return a message similar to the following example result, which gets logged by default in Application Insights. The properties of the `Trigger Details` can be used to locate and capture additional insights around messages received per `PartitionId`, `Offset`, and `SequenceNumber`.

Example result of the sample query:

JSON

```
"message": Trigger Details: PartitionId: 26, Offset: 17194119200,  
EnqueueTimeUtc: 2020-11-03T02:14:01.7740000Z, SequenceNumber: 843572, Count:  
10,
```

⚠ Warning

The library for Azure Java Functions currently has an issue that prevents access to the `PartitionID` and the `PartitionContext` when using `EventHubTrigger`. [Learn more in this GitHub issue report](#).

Tracking message flow using a transaction ID with Application Insights

In Application Insights, we can view all the telemetry related to a particular transaction by doing a Transaction search query on the transaction's `Operation ID` value. This can be especially useful for capturing the percentile values of average times for messages as the transaction moves through the event stream pipeline.

The following screenshot shows an example Transaction search in the Application Insights interface. The desired `Operation ID` is entered in the query field, identified with a magnifying glass icon (and shown here outlined in a red box). At the bottom of the main pane, the `Results` tab shows matching events in sequential order. In each event entry, the `Operation ID` value is highlighted in dark blue for easy verification.

Samltbkendwinyfx | Transaction search

Application Insights

Search (Ctrl+F)

Refresh Reset View in Logs Feedback Help Open Classic Search

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Investigate

Application map

Smart Detection

Live Metrics

Transaction search

Availability

Failures

Performance

Troubleshooting guides (previ...

Monitoring

Alerts

Metrics

Logs

Workbooks

Usage

Users

Sessions

Events

Funnels

Local Time : 10/8 11:58 PM - 11/10 10:58 PM Event types = All selected

1c8c9d7073a00e4bbdc8f2e65670e46

5 total results between 10/8/2020, 11:58:40 PM and 11/10/2020, 10:58:40 PM

6

4

2

0

Fri 9 Sun 11 Tue 13 Thu 15 Sat 17 Mon 19 Wed 21 Fri 23 Sun 25 Tue 27 Thu 29 Sat 31 Sun 1 Tue 3 Thu 5 Sat 7 Mon 9

11:58 PM

Availability Request Exception Page View Trace Custom Event Dependency

0 1 0 0 4 0 0

Results Grouped results. Loading... 0 loaded

Sort by time Newest first Oldest first

11/9/2020, 7:02:01 PM - TRACE Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succeeded, Id=b6d63bdd-2d20-461f-94f9-4fd46567ce72, Duration=28ms) Severity level: Information Operation Id: 1c8c9d7073a00e4bbdc8f2e65670e46

11/9/2020, 7:02:01 PM - TRACE Read Payload from message: {"Body":123,34,107,101,121,34,58,34,56,57,102,97,56,102,57,101,45,55,56,98,57,45,52,98,101,50,45,97,97,53,100,45,51,56,101,54,53,101,51,101,57,98,97,52,34} Severity level: Information Operation Id: 1c8c9d7073a00e4bbdc8f2e65670e46

11/9/2020, 7:02:01 PM - TRACE Trigger Details: PartitionId: 25, Offset: 30084705888, EnqueueTimeUtc: 2020-11-10T02:43:11.6580000Z, SequenceNumber: 1447822, Count: 10 Severity level: Information Operation Id: 1c8c9d7073a00e4bbdc8f2e65670e46

11/9/2020, 7:02:01 PM - TRACE Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Reason='(null)', Id=b6d63bdd-2d20-461f-94f9-4fd46567ce72) Severity level: Information Operation Id: 1c8c9d7073a00e4bbdc8f2e65670e46

11/9/2020, 7:02:01 PM - REQUEST EventHubStandardTriggeredFunction_AsSingleEvent Request URL: <empty> Response code: 0 Response time: 28.44 ms Operation Id: 1c8c9d7073a00e4bbdc8f2e65670e46

A query generated for a specific operation ID will look like the following. Note that the `Operation ID` GUID is specified in the third line's `where *` has clause. This example further narrows the query between two different `datetimes`.

Kusto

```
union isfuzzy=true availabilityResults, requests, exceptions, pageViews, traces, customEvents, dependencies
| where timestamp > datetime("2020-10-09T06:58:40.024Z") and timestamp < datetime("2020-11-11T06:58:40.024Z")
| where * has "1c8c9d7073a00e4bbdcc8f2e6570e46"
| order by timestamp desc
| take 100
```

Here is a screenshot of what the query and its matching results might look like in the Application Insights interface:

```
1 union isfuzzy=true availabilityResults, requests, exceptions, pageViews, traces, customEvents, dependencies
2 | where timestamp > datetime("2020-10-09T06:58:40.024Z") and timestamp < datetime("2020-11-11T06:58:40.024Z")
3 | where * has "1c8c9d7073a00e4bbdcc8f2e65670e46"
4 | order by timestamp desc
5 | take 100
```

Results Chart | Columns | Display time (UTC+00:00) | Group columns

Completed 00:06.2 5 records

timestamp [UTC]	id	name	success	message	duration	perform
11/10/2020, 3:02:01.037 AM	66c6044ed1be1040	EventHubStandardTriggeredFunction_AsSingleEvent	True		28.436	<250ms
11/10/2020, 3:02:01.037 AM				Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Re...		
11/10/2020, 3:02:01.037 AM				Trigger Details: PartitionId: 25, Offset: 30084705888, EnqueueTimeUtc:...		
11/10/2020, 3:02:01.065 AM				Read Payload from message: {"Body":123,34,107,101,121,34,58,34,56,...		
11/10/2020, 3:02:01.065 AM				Executed 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succ...		



Capture custom metrics from Azure Functions

.NET functions

Structured logging is used in the .NET Azure functions for capturing custom dimensions in the Application Insights traces table. These custom dimensions can then be used for querying data.

As an example, here is the log statement in the .NET `TransformingFunction`:

C#

```
log.LogInformation("TransformingFunction: Processed sensorDataJson=\n{sensorDataJson}, " +\n    "partitionId={partitionId}, offset={offset} at {enqueuedTimeUtc}, " +\n    "inputEH_enqueueTime={inputEH_enqueueTime}, processedTime=\n{processedTime}, " +\n    "transformingLatencyInMs={transformingLatencyInMs},\nprocessingLatencyInMs={processingLatencyInMs}",\n    sensorDataJson,\n    partitionId,\n    offset,\n    enqueuedTimeUtc,\n    inputEH_enqueueTime,\n    processedTime,\n    transformingLatency,\n    processingLatency);
```

The resulting logs created on Application Insights contain the above parameters as custom dimensions, as shown in this screenshot:

2/5/2021, 6:16:13.763 AM	TransformingFunction: Processed sensorDataJson={"Value":"eyJsb2N... 1	trace	{"prop_...
timestamp [UTC]	2021-02-05T06:16:13.7633341Z		
message	TransformingFunction: Processed sensorDataJson={"Value":"eyJsb2NhdGlvbil6IINiYXR0bGUiLCJyZWFlkaW5nljozMS4yLCJjYX		
severityLevel	1		
itemType	trace		
customDimensions	{"prop_{OriginalFormat}":"TransformingFunction: Processed sensorDataJson={sensorDataJson}, partitionId={partitionId}, offset={offset}, processedTime={processedTime}, processingLatency={processingLatency}, transformingLatency={transformingLatency}, inputEH_enqueuedTime={inputEH_enqueuedTime}, enqueuedTimeUtc={enqueuedTimeUtc}, offset={offset}, partitionId={partitionId}, processedTime={processedTime}, processingLatency={processingLatency}, transformingLatency={transformingLatency}, sensorDataJson={sensorDataJson}"}		
Category	Function.TransformingFunction.User		
HostInstanceId	63f8de3b-42c9-4cf8-8585-c62ff9d7dfb8		
InvocationId	b24e2f69-90e9-4c4d-a20f-70ea23f3a051		
LogLevel	Information		
ProcessId	10088		
prop_Links	System.Collections.Generic.List`1[System.Diagnostics.Activity]		
prop_enqueuedTimeUtc	2021-02-05T06:16:13.6590000Z		
prop_inputEH_enqueuedTime	2021-02-05T06:16:13.6390000Z		
prop_offset	11846672		
prop_partitionId	22		
prop_processedTime	2021-02-05T06:16:13.7630000Z		
prop_processingLatency	00:00:00.1242571		
prop_sensorDataJson	{"Value":"eyJsb2NhdGlvbil6IINiYXR0bGUiLCJyZWFlkaW5nljozMS4yLCJjYXRIZ29yeSl6InRibXBlcmF0dXJli		
prop_transformingLatency	00:00:00.1042571		
prop_{OriginalFormat}	TransformingFunction: Processed sensorDataJson={sensorDataJson}, partitionId={partitionId}, offset={offset}, processedTime={processedTime}, processingLatency={processingLatency}, transformingLatency={transformingLatency}, sensorDataJson={sensorDataJson}"}		

These logs can be queried as follows:

```
Kusto

traces
| where timestamp between(min_t .. max_t)
// Function name should be of the function consuming from the Event Hub of
interest
| where operation_Name == "{Function_Name}"
| where message has "{Function_Name}: Processed"
| project timestamp = todatetime(customDimensions.prop__enqueuedTimeUtc)
```

! Note

In order to make sure we do not affect performance in these tests, we have turned on the sampling settings of Azure Function logs for Application Insights using the `host.json` file as shown below. This means that all statistics captured from logging are considered to be average values and not actual counts.

host.json example:

```
JSON
"logging": {
  "applicationInsights": {
```

```

        "samplingExcludedTypes": "Request",
        "samplingSettings": {
            "isEnabled": true
        }
    }
}

```

Java functions

Currently, structured logging isn't supported in Java Azure functions for capturing custom dimensions in the Application Insights traces table.

As an example, here is the log statement in the Java `TransformingFunction`:

Java

```

LoggingUtilities.logSuccessInfo(
    context.getLogger(),
    "TransformingFunction",
    "SuccessInfo",
    offset,
    processedTimeString,
    dateFormatter.format(enqueuedTime),
    transformingLatency
);

```

The resulting logs created on Application Insights contain the above parameters in the message as shown below:

2/25/2021, 11:44:05.958 PM	{"functionName": "TransformingFunction", "logType": "SuccessInfo", "..."}	1	trace	{"HostInstanceId": "bd6b9e0b-4e58-4b21-b7a1-e906feaec502", "InvocationId": "b1ac3952-345a-4b83-bcaf-9fe4e5f8984d", "LogLevel": "Information", "Category": "Function.TransformingFunction.User", "ProcessId": "5832"}	Transfo...
timestamp [UTC]	2021-02-25T23:44:05.958379Z				
message	{"functionName": "TransformingFunction", "logType": "SuccessInfo", "offset": "17189587000", "processedTime": "2021-02-25T23:44:05.930251Z", "enqueuedTime": "2021-02-25T23:19:42.018Z", "latency": "2393233", "partitionKey": "null"}				
enqueuedTime	2021-02-25T23:19:42.018Z				
functionName	TransformingFunction				
latency	2393233				
logType	SuccessInfo				
offset	17189587000				
partitionKey	null				
processedTime	2021-02-25T23:44:05.930251Z				
severityLevel	1				
itemType	trace				
customDimensions	{"HostInstanceId": "bd6b9e0b-4e58-4b21-b7a1-e906feaec502", "InvocationId": "b1ac3952-345a-4b83-bcaf-9fe4e5f8984d", "LogLevel": "Information", "Category": "Function.TransformingFunction.User", "ProcessId": "5832"}				

These logs can be queried as follows:

Kusto

```

traces
| where timestamp between(min_t .. max_t)
// Function name should be of the function consuming from the Event Hub of
interest
| where operation_Name in ("{$Function name}") and message contains

```

```
"SuccessInfo"
| project timestamp = todatetime(tostring(parse_json(message).enqueuedTime))
```

ⓘ Note

In order to make sure we do not affect performance in these tests, we have turned on the sampling settings of Azure Function logs for Application Insights using the `host.json` file as shown below. This means that all statistics captured from logging are considered to be average values and not actual counts.

`host.json` example:

JSON

```
"logging": {
  "applicationInsights": {
    "samplingExcludedTypes": "Request",
    "samplingSettings": {
      "isEnabled": true
    }
  }
}
```

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rajasa Savant](#) | Senior Software Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Related resources

- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the reference architecture work.

- [Private link scenario in event stream processing](#) is a solution idea for implementing a similar architecture in a virtual network (VNet) with private endpoints, in order to enhance security.
- [Azure Kubernetes in event stream processing](#) describes a variation of a serverless event-driven architecture running on Azure Kubernetes with KEDA scaler.

Serverless Functions app operations

Article • 05/06/2022

This article describes Azure operations considerations for serverless Functions applications. To support Functions apps, operations personnel need to:

- Understand and implement hosting configurations.
- Future-proof scalability by automating infrastructure provisioning.
- Maintain business continuity by meeting availability and disaster recovery requirements.

Planning

To plan operations, understand your workloads and their requirements, then design and configure the best options for the requirements.

Choose a hosting option

The Azure Functions Runtime provides flexibility in hosting. Use the [hosting plan comparison table](#) to determine the best choice for your requirements.

- Azure Functions hosting plans

Each Azure Functions project deploys and runs in its own Functions app, which is the unit of scale and cost. The three hosting plans available for Azure Functions are the Consumption plan, Premium plan, and Dedicated (App Service) plan. The hosting plan determines scaling behavior, available resources, and support for advanced features like virtual network connectivity.

- Azure Kubernetes Service (AKS)

Kubernetes-based Functions provides the Functions Runtime in a Docker container with event-driven scaling through Kubernetes-based Event Driven Autoscaling (KEDA).

For more information about hosting plans, see:

- [Azure Functions scale and hosting](#)
- [Consumption plan](#)
- [Premium plan](#)
- [Dedicated \(App Service\) plan](#)
- [Azure Functions on Kubernetes with KEDA](#)

- Azure subscription and service limits, quotas, and constraints

Understand scaling

The serverless Consumption and Premium hosting plans *scale* automatically, adding and removing Azure Functions host instances based on the number of incoming events. Scaling can vary on several dimensions, and behave differently based on plan, trigger, and code language.

For more information about scaling, see:

- [Understand scaling behaviors](#)
- [Scalability best practices](#)

Understand and address cold starts

If the number of host instances scales down to zero, the next request has the added latency of restarting the Function app, called a *cold start*. [Cold start](#) is a large discussion point for serverless architectures, and a point of ambiguity for Azure Functions.

The Premium hosting plan prevents cold starts by keeping some instances warm. Reducing dependencies and using asynchronous operations in the Functions app also minimizes the impact of cold starts. However, availability requirements may require running the app in a Dedicated hosting plan with *Always on* enabled. The Dedicated plan uses dedicated virtual machines (VMs), so is not serverless.

For more information about cold start, see [Understanding serverless cold start](#).

Identify storage considerations

Every Azure Functions app relies on Azure Storage for operations such as managing triggers and logging function executions. When creating a Functions app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. For more information, see [Storage considerations for Azure Functions](#).

Identify network design considerations

Networking options let the Functions app restrict access, or access resources without using internet-routable addresses. The hosting plans offer different levels of network isolation. Choose the option that best meets your network isolation requirements. For more information, see [Azure Functions networking options](#).

Production

To prepare the application for production, make sure you can easily redeploy the hosting plan, and apply scale-out rules.

Automate hosting plan provisioning

With infrastructure as code, you can automate infrastructure provisioning. Automatic provisioning provides more resiliency during disasters, and more agility to quickly redeploy the infrastructure as needed.

For more information on automated provisioning, see:

- [Automate resource deployment for your function app in Azure Functions](#)
- [Terraform - Manages a Function App ↗](#)

Configure scale out options

Autoscale provides the right amount of running resources to handle application load. Autoscale adds resources to handle increases in load, and saves money by removing resources that are idle.

For more information about autoscale options, see:

- [Premium Plan settings](#)
- [App Service Plan settings](#)

Optimization

When the application is in production, make sure that:

- The hosting plan can scale to meet application demands.
- There's a plan for business continuity, availability, and disaster recovery.
- You can monitor hosting and application health and receive alerts.

Implement availability requirements

Azure Functions run in a specific region. To get higher availability, you can deploy the same Functions app to multiple regions. In multiple regions, Functions can run in the *active-active* or *active-passive* availability pattern.

For more information about Azure Functions availability and disaster recovery, see:

- [Azure Functions geo-disaster recovery](#)
- [Disaster recovery and geo-distribution in Azure Durable Functions](#)

Monitoring logging, application monitoring, and alerting

Application Insights and logs in Azure Monitor automatically collect log, performance, and error data and detect performance anomalies. Azure Monitor includes powerful analytics tools to help diagnose issues and understand function use. Application Insights help you continuously improve performance and usability.

For more information about monitoring and analyzing Azure Functions performance, see:

- [Monitor Azure Functions](#)
- [Monitor Azure Functions with Azure Monitor logs](#)
- [Application Insights for Azure Functions supported features](#)

Next steps

- [Serverless application development and deployment](#)
- [Azure Functions app security](#)

Serverless Functions security

Article • 10/13/2022

This article describes Azure services and activities security personnel can implement for serverless Functions. These guidelines and resources help develop secure code and deploy secure applications to the cloud.

Planning

The primary goals of a secure serverless Azure Functions application environment are to protect running applications, quickly identify and address security issues, and prevent future similar issues.

The [OWASP Serverless Top 10](#) describes the most common serverless application security vulnerabilities, and provides basic techniques to identify and protect against them.

In many ways, planning for secure development, deployment, and operation of serverless functions is much the same as for any web-based or cloud hosted application. Azure App Service provides the hosting infrastructure for your function apps. [Securing Azure Functions](#) article provides security strategies for running your function code, and how App Service can help you secure your functions.

For more information about Azure security, best practices, and shared responsibilities, see:

- [Security in Azure App Service](#)
- [Built-in security controls](#)
- [Secure development best practices on Azure](#).
- [Security best practices for Azure solutions \(PDF report\)](#)
- [Shared responsibilities for cloud computing \(PDF report\)](#)

Deployment

To prepare serverless Functions applications for production, security personnel should:

- Conduct regular code reviews to identify code and library vulnerabilities.
- Define resource permissions that Functions needs to execute.
- Configure network security rules for inbound and outbound communication.
- Identify and classify sensitive data access.

The [Azure Security Baseline for Azure Functions](#) article contains more recommendations that will help you improve the security posture of your deployment.

Keep code secure

Find security vulnerabilities and errors in code and manage security vulnerabilities in projects and dependencies.

For more information, see:

- [GitHub - Finding security vulnerabilities and errors in your code ↗](#)
- [GitHub - Managing security vulnerabilities in your project ↗](#)

Perform input validation

Different event sources like Blob storage, Azure Cosmos DB NoSQL databases, event hubs, queues, or Graph events can trigger serverless Functions. Injections aren't strictly limited to inputs coming directly from the API calls. Functions may consume other input from the possible event sources.

In general, don't trust input or make any assumptions about its validity. Always use safe APIs that sanitize or validate the input. If possible, use APIs that bind or parameterize variables, like using prepared statements for SQL queries.

For more information, see:

- [Azure Functions Input Validation with FluentValidation ↗](#)
- [Security Frame: Input Validation Mitigations](#)
- [HTTP Trigger Function Request Validation ↗](#)
- [How to validate request for Azure Functions ↗](#)

Secure HTTP endpoints for development, testing, and production

Azure Functions lets you use keys to make it harder to access your HTTP function endpoints. To fully secure your function endpoints in production, consider implementing one of the following Function app-level security options:

- Turn on App Service authentication and authorization for your Functions app. See [Authorization keys](#).
- Use Azure API Management (APIM) to authenticate requests. See [Import an Azure Function App as an API in Azure API Management](#).

- Deploy your Functions app to an Azure App Service Environment (ASE).
- Use an App Service Plan that restricts access, and implement Azure Front Door + WAF to handle your incoming requests. See [Create a Front Door for a highly available global web application](#).

For more information, see [Secure an HTTP endpoint in production](#).

Set up Azure role-based access control (Azure RBAC)

Azure role-based access control (Azure RBAC) has several Azure built-in roles that you can assign to users, groups, service principals, and managed identities to control access to Azure resources. If the built-in roles don't meet your organization's needs, you can create your own Azure custom roles.

Review each Functions app before deployment to identify excessive permissions. Carefully examine functions to apply "least privilege" permissions, giving each function only what it needs to successfully execute.

Use Azure RBAC to assign permissions to users, groups, and applications at a certain scope. The scope of a role assignment can be a subscription, a resource group, or a single resource. Avoid using wildcards whenever possible.

For more information about Azure RBAC, see:

- [What is Azure role-based access control \(Azure RBAC\)?](#)
- [Azure built-in roles](#)
- [Azure custom roles](#)

Use managed identities and key vaults

A common challenge when building cloud applications is how to manage credentials for authenticating to cloud services in your code. Credentials should never appear in application code, developer workstations, or source control. Instead, use a key vault to store and retrieve keys and credentials. Azure Key Vault provides a way to securely store credentials, secrets, and other keys. The code authenticates to Key Vault to retrieve the credentials.

For more information, see [Use Key Vault references for App Service and Azure Functions](#).

Managed identities let Functions apps access resources like key vaults and storage accounts without requiring specific access keys or connection strings. A full audit trail in the logs displays which identities execute requests to resources. Use Azure RBAC and

managed identities to granularly control exactly what resources Azure Functions applications can access.

For more information, see:

- [What are managed identities for Azure resources?](#)
- [How to use managed identities for App Service and Azure Functions](#)

Use shared access signature (SAS) tokens to limit access to resources

A *shared access signature (SAS)* provides secure delegated access to resources in your storage account, without compromising the security of your data. With a SAS, you have granular control over how a client can access your data. You can control what resources the client may access, what permissions they have on those resources, and how long the SAS is valid, among other parameters.

For more information, see [Grant limited access to Azure Storage resources using shared access signatures \(SAS\)](#).

Secure Blob storage

Identify and classify sensitive data, and minimize sensitive data storage to only what is necessary. For sensitive data storage, add multi-factor authentication and data encryption in transit and at rest. Grant limited access to Azure Storage resources using SAS tokens.

For more information, see [Security recommendations for Blob storage](#).

Optimization

Once an application is in production, security personnel can help optimize workflow and prepare for scaling.

Use Microsoft Defender for Cloud and apply security recommendations

Microsoft Defender for Cloud is a security scanning solution for your application that identifies potential security vulnerabilities and creates recommendations. The recommendations guide you to configure needed controls to harden and protect your resources.

For more information, see:

- [Protect your applications with Microsoft Defender for Cloud](#)
- [Defender for Cloud app recommendations](#)

Enforce application governance policies

Apply centralized, consistent enforcements and safeguards to your application at scale. For more information, see [Azure Policy built-in policy definitions](#).

Next steps

- [Serverless application development and deployment](#)
- [Azure Functions app operations](#)

Azure security baseline for Functions

Article • 09/20/2023

This security baseline applies guidance from the [Microsoft cloud security benchmark version 1.0](#) to Functions. 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 Functions.

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 Functions have been excluded. To see how Functions completely maps to the Microsoft cloud security benchmark, see the [full Functions security baseline mapping file](#).

Security profile

The security profile summarizes high-impact behaviors of Functions, which may result in increased security considerations.

Service Behavior Attribute	Value
Product Category	Compute, Web
Customer can access HOST / OS	No Access
Service can be deployed into customer's virtual network	True
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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Deploy the service into a virtual network. Assign private IPs to the resource (where applicable) unless there is a strong reason to assign public IPs directly to the resource.

Note: Networking features are exposed by the service but need to be configured for the application. By default, public network access is allowed.

Reference: [Azure Functions networking options](#)

Network Security Group Support

Description: Service network traffic respects Network Security Groups rule assignment on its subnets. [Learn more](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use network security groups (NSG) to restrict or monitor traffic by port, protocol, source IP address, or destination IP address. Create NSG rules to restrict your service's open ports (such as preventing management ports from being accessed from untrusted networks). Be aware that by default, NSGs deny all inbound traffic but allow traffic from virtual network and Azure Load Balancers.

Reference: [Azure Functions networking options](#)

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Deploy private endpoints for all Azure resources that support the Private Link feature, to establish a private access point for the resources.

Reference: [Azure Functions networking options](#)

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Azure Functions can be configured with private endpoints, but there is not presently a single toggle for disabling public network access absent configuring private endpoints.

Configuration Guidance: Disable public network access either using the service-level IP ACL filtering rule or a toggling switch for public network access.

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Customer-owned endpoints may be configured to require Azure AD authentication requirements. System-provided endpoints for deployment operations and advanced developer tools support Azure AD but by default have the ability to alternatively use publishing credentials. These publishing credentials can be disabled. Some data plane endpoints on the app may be accessed by administrative keys configured in the Functions host, and these are not configurable with Azure AD requirements at this time.

Configuration Guidance: Use Azure Active Directory (Azure AD) as the default authentication method to control your data plane access.

Reference: [Configure deployment credentials - disable basic authentication](#)

Local Authentication Methods for Data Plane Access

Description: Local authentication methods supported for data plane access, such as a local username and password. [Learn more.](#)

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Feature notes: Deployment credentials are created by default, but they can be disabled. Some operations exposed by the application runtime may be performed using an administrative key, which cannot presently be disabled. This key can be stored in Azure Key Vault, and it can be regenerated at any time. Avoid the usage of local authentication methods or accounts, these should be disabled wherever possible. Instead use Azure AD to authenticate where possible.

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: [Disable basic authentication](#)

IM-3: Manage application identities securely and automatically

Features

Managed Identities

Description: Data plane actions support authentication using managed identities. [Learn more.](#)

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: [How to use managed identities for App Service and Azure Functions](#)

Service Principals

Description: Data plane supports authentication using service principals. [Learn more.](#)

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.

Microsoft Defender for Cloud monitoring

Azure Policy built-in definitions - Microsoft.Web:

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
App Service apps should	Use a managed identity for	AuditIfNotExists,	3.0.0 ↗

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
use managed identity ↗	enhanced authentication security	Disabled	
Function apps should use managed identity ↗	Use a managed identity for enhanced authentication security	AuditIfNotExists, Disabled	3.0.0 ↗

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: For data plane endpoints which are not defined by the application, conditional access would need to be configured against Azure Service Management.

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Ensure that secrets and credentials are stored in secure locations such as Azure Key Vault, instead of embedding them into code or configuration files.

Reference: [Use Key Vault references for App Service and Azure Functions](#)

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](#).

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: The only data-plane actions which can leverage Azure RBAC are the Kudu/SCM/deployment endpoints. These require permission over the `Microsoft.Web/sites/publish/Action` operation. Endpoints exposed by the customer application itself are not covered by Azure RBAC.

Configuration Guidance: Use Azure role-based access control (Azure RBAC) to manage Azure resource access through built-in role assignments. Azure RBAC roles can be assigned to users, groups, service principals, and managed identities.

Reference: [RBAC permissions required to access Kudu](#)

PA-8: Determine access process for cloud provider support

Features

Customer Lockbox

Description: Customer Lockbox can be used for Microsoft support access. [Learn more.](#)

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-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](#).

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Function apps are created by default to support TLS 1.2 as a minimum version, but an app can be configured with a lower version through a configuration setting. HTTPS is not required of incoming requests by default, but this can also be set via a configuration setting, at which point any HTTP request will be automatically redirected to use HTTPS.

Configuration Guidance: Enable secure transfer in services where there is a native data in transit encryption feature built in. Enforce HTTPS on any web applications and services and ensure TLS v1.2 or later is used. Legacy versions such as SSL 3.0, TLS v1.0 should be disabled. For remote management of Virtual Machines, use SSH (for Linux) or RDP/TLS (for Windows) instead of an unencrypted protocol.

Reference: [Add and manage TLS/SSL certificates in Azure App Service](#)

Microsoft Defender for Cloud monitoring

Azure Policy built-in definitions - Microsoft.Web:

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
App Service apps should only be accessible over HTTPS ↗	Use of HTTPS ensures server/service authentication and protects data in transit from network layer eavesdropping attacks.	Audit, Disabled, Deny	4.0.0 ↗
App Service apps should require FTPS only ↗	Enable FTPS enforcement for enhanced security.	AuditIfNotExists, Disabled	3.0.0 ↗
App Service apps should use the latest TLS version ↗	Periodically, newer versions are released for TLS either due to security flaws, include additional functionality, and enhance speed. Upgrade to the latest TLS version for App Service apps to take advantage of security fixes, if any, and/or new functionalities of the latest version.	AuditIfNotExists, Disabled	2.0.1 ↗
Function apps should only be accessible over HTTPS ↗	Use of HTTPS ensures server/service authentication and protects data in transit from network layer eavesdropping attacks.	Audit, Disabled, Deny	5.0.0 ↗
Function apps should require FTPS only ↗	Enable FTPS enforcement for enhanced security.	AuditIfNotExists, Disabled	3.0.0 ↗
Function apps should use the latest TLS version ↗	Periodically, newer versions are released for TLS either due to security flaws, include additional functionality, and enhance speed. Upgrade to the latest TLS version for Function apps to take advantage of security fixes, if any, and/or new functionalities of the latest version.	AuditIfNotExists, Disabled	2.0.1 ↗

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Azure Functions does not directly support this feature, but an application can be configured to leverage services which do, in place of any possible data storage in Functions. Azure Files may be mounted as the file system, all App Settings, including secrets, may be stored in Azure Key Vault, and deployment options such as run-from-package may pull content from Azure Blob storage.

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: [Encrypt your application data at rest using customer-managed keys](#)

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](#).

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: [Use Key Vault references for App Service and Azure Functions](#)

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure Key Vault to create and control the certificate lifecycle, including creation, importing, rotation, revocation, storage, and purging of the certificate. Ensure the certificate generation follows defined standards without using any insecure properties, such as: insufficient key size, overly long validity period, insecure cryptography. Setup automatic rotation of the certificate in Azure Key Vault and the Azure service (if supported) based on a defined schedule or when there is a certificate expiration. If automatic rotation is not supported in the application, ensure they are still rotated using manual methods in Azure Key Vault and the application.

Reference: [Add a TLS/SSL certificate in Azure App 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.](#)

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.

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Defender for App Service includes Azure Functions. If this solution is enabled, function apps under the enablement scope will be included.

Configuration Guidance: Use Azure Active Directory (Azure AD) as the default authentication method to control your management plane access. When you get an alert from Microsoft Defender for Key Vault, investigate and respond to the alert.

Reference: [Defender for App 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](#).

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Enable resource logs for the service. For example, Key Vault supports additional resource logs for actions that get a secret from a key vault or and Azure SQL has resource logs that track requests to a database. The content of resource logs varies by the Azure service and resource type.

Reference: [Monitoring Azure Functions with Azure Monitor Logs](#)

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.](#)

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Feature notes: A feature for backing up an application is available if hosted on a Standard, Premium, or Isolated App Service plan. This feature does not leverage Azure Backup and does not include event sources or externally linked storage. See [/azure/app-service/manage-backup](#) for more details.

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.](#)

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: A backup feature is available to apps running on Standard, Premium, and Isolated App Service plans. This does not include backing up event sources or externally provided storage.

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.

Reference: [Back up and restore your app in Azure App Service](#)

Next steps

- See the [Microsoft cloud security benchmark overview](#)
- Learn more about [Azure security baselines](#)

Azure serverless overview: Create cloud-based apps and solutions with Azure Logic Apps and Azure Functions

Article • 01/04/2024

While serverless doesn't mean "no servers", Azure serverless helps you spend less on managing your infrastructure. In traditional app development, you can spend much time and energy on discussing and addressing hosting, scaling, and monitoring solutions to meet your app requirements and demands. With serverless apps and solutions, you can more easily handle these concerns as part of the app or solution. Serverless offers other benefits such as faster development, less code, simplicity, and scaling flexibility. All these capabilities free you to focus more on the business logic. Also, serverless is typically billed or charged based on usage. So, if no consumption happens, no charges are incurred. For more information, learn more about [Azure serverless](#).

This article briefly summarizes the core serverless offerings in Azure, which are Azure Logic Apps and Azure Functions. Both services align with the previously described principles and help you build robust cloud apps and solutions with minimal code.

For more introductory information, visit the Azure pages for [Azure Logic Apps](#) and [Azure Functions](#). For more detailed information, review the documentation pages for [What is Azure Logic Apps](#) and [What is Azure Functions](#).

Azure Logic Apps

This service provides simplified ways to design, develop, and orchestrate automated event-driven integration workflows that run and scale in the cloud. With Azure Logic Apps, you can use a visual designer to quickly model business processes as workflows. A workflow always starts with a trigger as the first step. Following the trigger, one or more actions run subsequent operations in the workflow. These operations can include various combinations of actions, including conditional logic and data conversions.

To connect your workflows to other Azure services, Microsoft services, cloud-based environments, and on-premises environments without writing any code, you can add prebuilt triggers and actions to your workflows by choosing from [hundreds of connectors](#), all managed by Microsoft. Each connector is actually a proxy or wrapper around an API, which lets the underlying service communicate with Azure Logic Apps. For example, the Office 365 Outlook connector offers a trigger named **When a new email arrives**. For serverless apps and solutions, you can use Azure Logic Apps to

orchestrate multiple functions created in Azure Functions. By doing so, you can easily call various functions as a single process, especially when the process requires working with an external API or system.

If no connector is available to meet your needs, you can use the built-in HTTP operation or Request trigger to communicate with any service endpoint. Or, you can create your own connector using an existing API.

Based on the logic app resource type that you choose, the associated workflow runs in either multi-tenant Azure Logic Apps, single-tenant Azure Logic Apps, or a dedicated integration service environment (ISE). Each has their own capabilities, benefits, and billing models. The Azure portal provides the fastest way to get started creating logic app workflows. However, you can also use other tools such as Visual Studio Code, Visual Studio, Azure PowerShell, and others. For more information, review [What is Azure Logic Apps?](#)

To get started with Azure Logic Apps, try a [quickstart to create an example Consumption logic app workflow in multi-tenant Azure Logic Apps using the Azure portal](#). Or, try these [steps that create an example serverless app with Azure Logic Apps and Azure Functions in Visual Studio](#).

For other information, review the following documentation:

- [What is Azure Logic Apps?](#)
- [Managed connectors for Azure Logic Apps](#)
- [Built-in connectors for Azure Logic Apps](#)
- [Single-tenant versus multi-tenant and integration service environment for Azure Logic Apps](#)
- [Usage metering, billing, and pricing models for Azure Logic Apps](#)

Azure Functions

This service provides a simplified way to write and run pieces of code or *functions* in the cloud. You can write only the code you need for the current problem, without setting up a complete app or the required infrastructure, which makes development faster and more productive. Use your chosen development language, such as C#, Java, JavaScript, PowerShell, Python, and TypeScript. You're billed only for the duration when your code runs, and Azure scales as necessary.

To get started with Azure Functions, try [creating your first Azure function in the Azure portal](#).

For other information, review the following documentation:

- [What is Azure Functions?](#)
- [Getting started with Azure Functions](#)
- [Supported languages in Azure Functions](#)
- [Azure Functions hosting options](#)
- [Azure Functions pricing](#)

Get started with serverless apps in Azure

Azure provides rich tools for developing, deploying, and managing serverless apps. You can create serverless apps using the Azure portal, Visual Studio, or [Visual Studio Code](#). After you build your app, you can [deploy that app quickly with Azure Resource Manager templates](#). Azure also provides monitoring, which you can access through the Azure portal, through the API or SDKs, or with integrated tooling for Azure Monitor logs and Application Insights.

Next steps

- [Create an example serverless app with Azure Logic Apps and Azure Functions in Visual Studio](#)
- [Create a customer insights dashboard with serverless](#)

Integrate Event Hubs with serverless functions on Azure

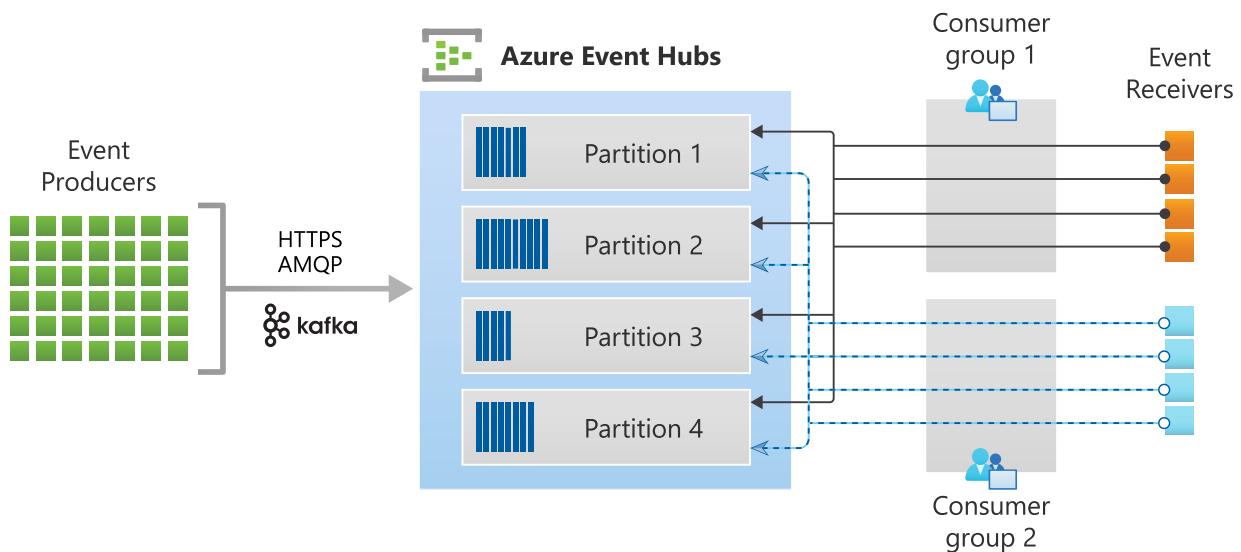
Azure Event Hubs Azure Functions Azure Monitor

Solutions that use Azure Event Hubs together with Azure Functions benefit from a [serverless](#) architecture that is scalable, cost-effective, and capable of processing large volumes of data in near real time. As much as these services work seamlessly together, there are many features, settings, and intricacies that add complexity to their relationship. This article provides guidance on how to effectively take advantage of this integration by highlighting key considerations and techniques for performance, resiliency, security, observability, and scale.

Event Hubs core concepts

[Azure Event Hubs](#) is a highly scalable event processing service that can receive millions of events per second. Before diving into the patterns and best practices for Azure Functions integration, it's best to understand the fundamental components of Event Hubs.

The following diagram shows the Event Hubs stream processing architecture:

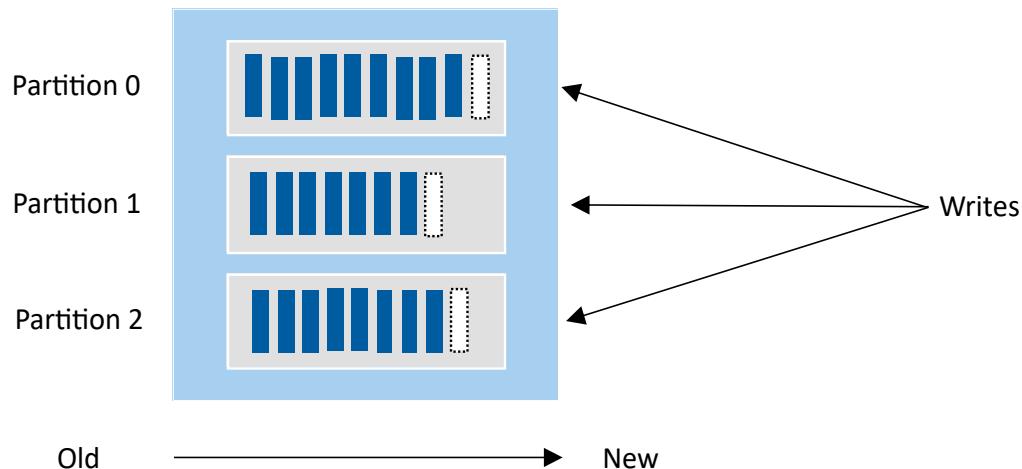


Events

An event is a notification or state change that is represented as a fact that happened in the past. Events are immutable and persisted in an **event hub**, also referred to as a *topic* in [Kafka](#). An event hub is comprised of one or more **partitions**.

Partitions

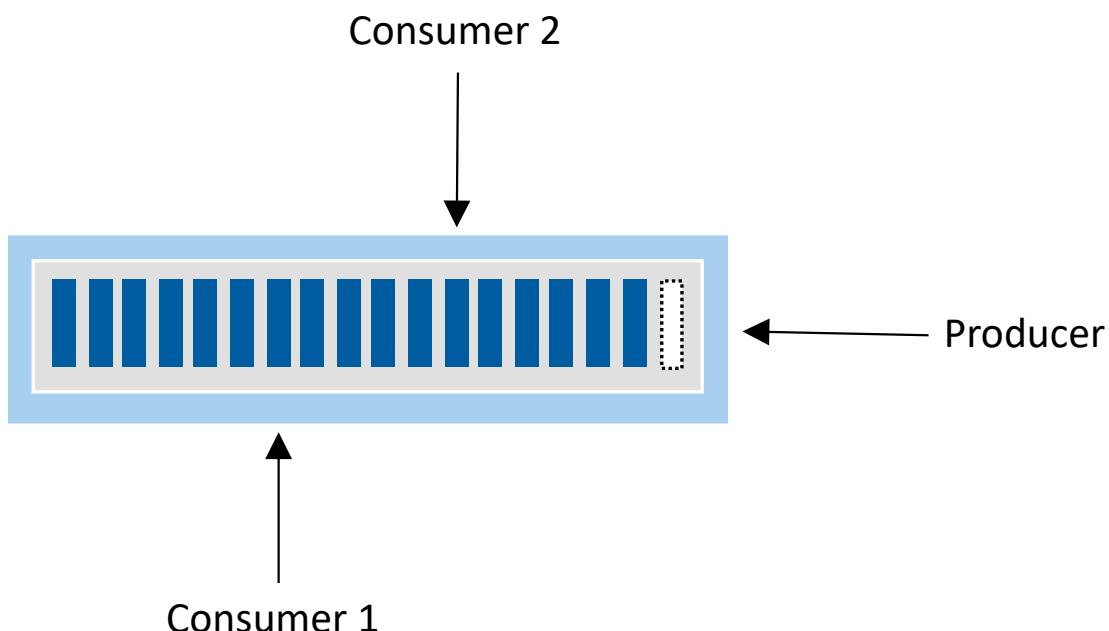
When a partition isn't specified by the sender, received events are distributed across partitions in the event hub. Each event is written in exactly one partition and isn't multi-cast across partitions. Each partition works as a log where records are written in an append-only pattern. The analogy of a *commit log* is frequently used to describe the nature of how events are added to the end of a sequence in a partition.



When more than one partition is used, it allows for parallel logs to be used from within the same event hub. This behavior provides multiple degrees of parallelism and enhances throughput for consumers.

Consumers and consumer groups

A partition can be consumed by more than one consumer, each reading from and managing their own offsets.



Event Hubs has the concept of [consumers groups](#), which enables multiple consuming applications to each have a separate view of the event stream and read the stream independently at their own pace and with their own offsets.

To learn more, see [Deep dive on Event Hubs concepts and features](#).

Consuming events with Azure Functions

Azure Functions supports [trigger](#) and [output](#) bindings for Event Hubs. This section covers how Azure Functions responds to events sent to an event hub event stream using triggers.

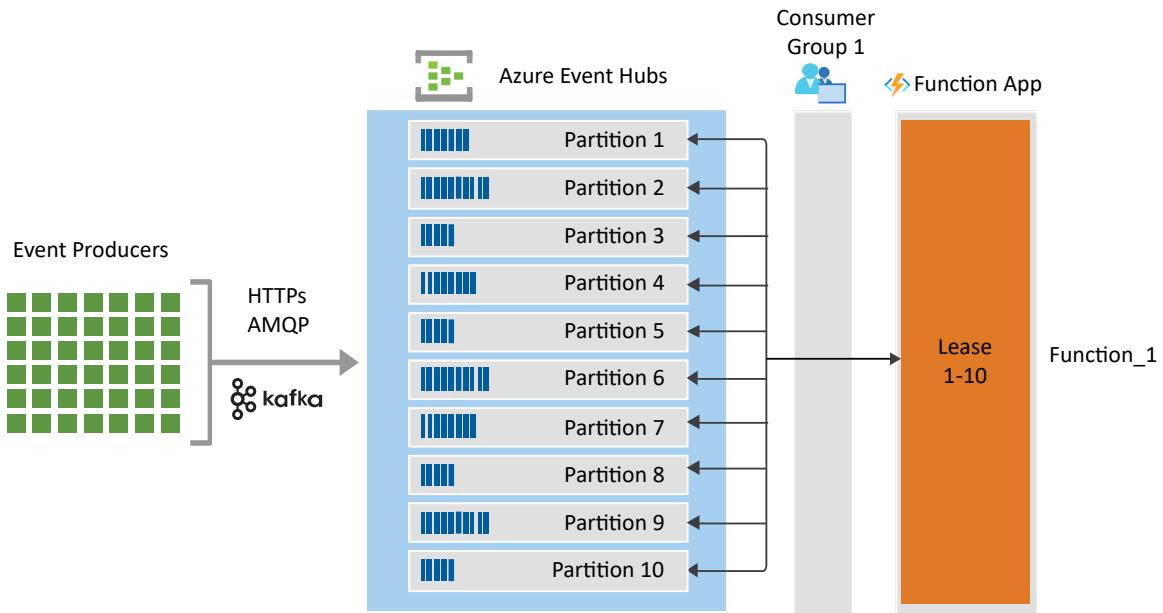
Each instance of an Event Hubs triggered function is backed by a single [EventProcessorHost](#) instance. The trigger (powered by Event Hubs) ensures that only one [EventProcessorHost](#) instance can get a lease on a given partition.

For example, consider an event hub with the following characteristics:

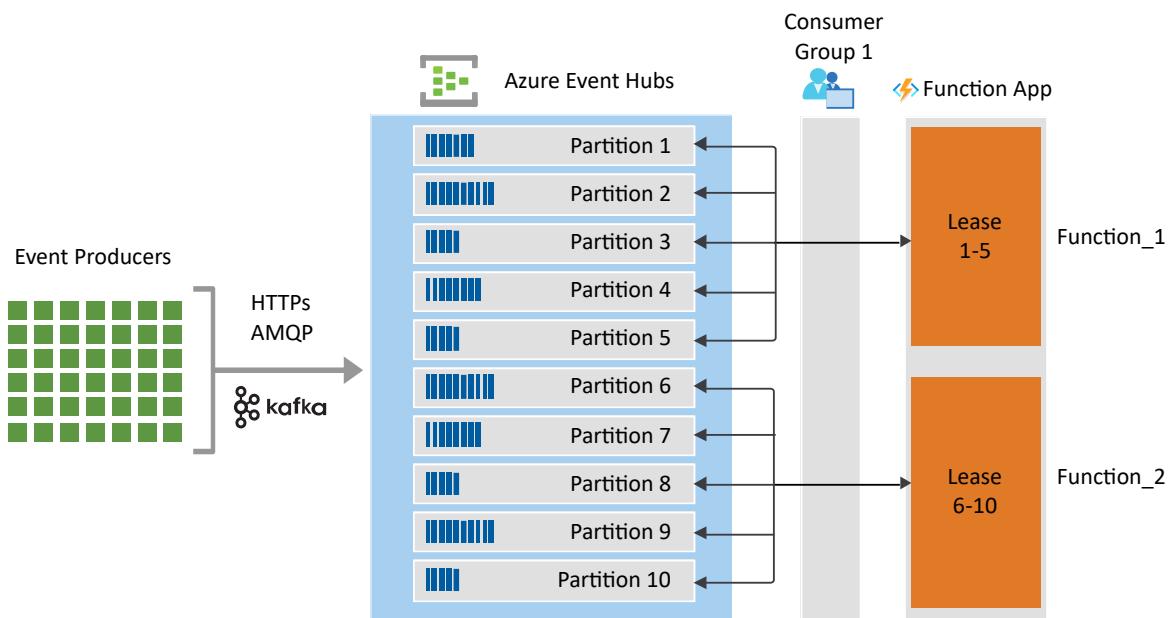
- 10 partitions.
- 1,000 events distributed evenly all partitions, with a varying number of messages in each partition.

When your function is first enabled, there's only one instance of the function. Let's call the first function instance `Function_1`. `Function_1` has a single instance of [EventProcessorHost](#) that holds a lease on all 10 partitions. This instance is reading events from partitions 1-10. From this point forward, one of the following happens:

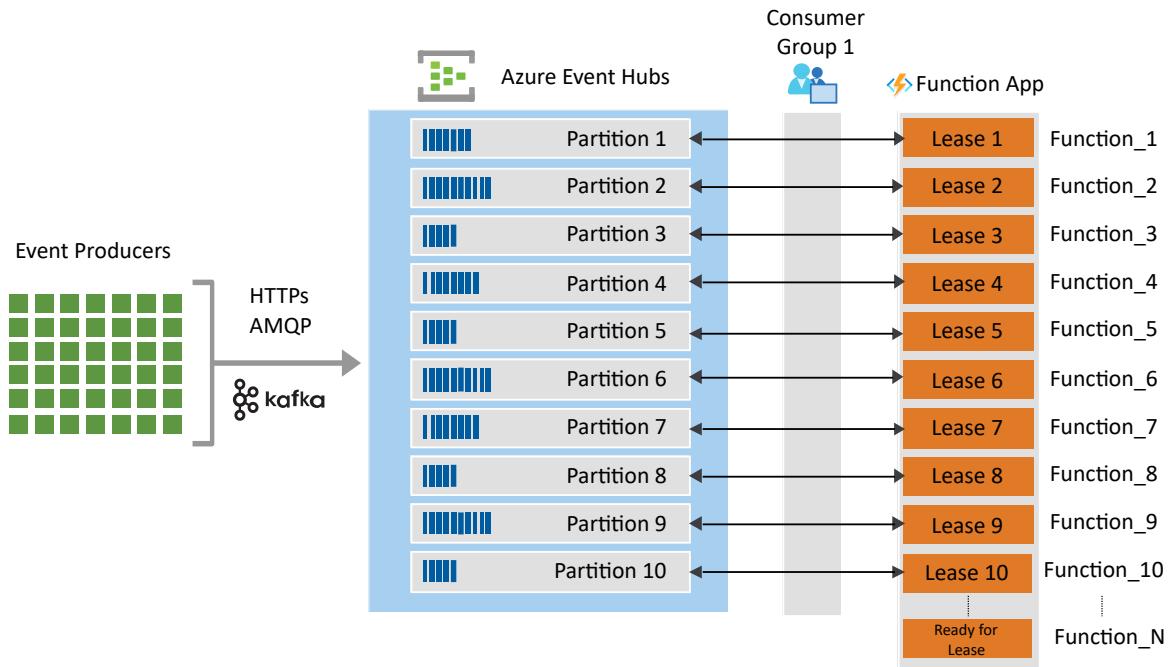
- **New function instances are not needed:** `Function_1` can process all 1,000 events before the Functions scaling logic take effect. In this case, all 1,000 messages are processed by `Function_1`.



- **An additional function instance is added:** event-based scaling or other automated or manual logic might determine that `Function_1` has more messages than it can process and then creates a new function app instance (`Function_2`). This new function also has an associated instance of `EventProcessorHost`. As the underlying event hub detects that a new host instance is trying read messages, it load balances the partitions across the host instances. For example, partitions 1-5 may be assigned to `Function_1` and partitions 6-10 to `Function_2`.



- **N more function instances are added:** event-based scaling or other automated or manual logic determines that both `Function_1` and `Function_2` have more messages than they can process, new `Function_N` function app instances are created. Instances are created to the point where N is equal to or greater than the number of event hub partitions. In our example, Event Hubs again load balances the partitions, in this case across the instances `Function_1` ... `Function_10`.



As scaling occurs, N instances can be a number greater than the number of event hub partitions. This situation might occur while event-driven scaling stabilizes instance counts, or because other automated or manual logic created more instances than partitions. In this case, [EventProcessorHost](#) instances will only obtain locks on partitions as they become available from other instances, as at any given time only one function instance from the same consumer group can access/read from the partitions it has locks on.

When all function execution completes (with or without errors), checkpoints are added to the associated storage account. When check-pointing succeeds, all 1,000 messages are never retrieved again.

Dynamic, event-based scaling is possible with Consumption and Premium Azure plans. Kubernetes hosted function apps can also take advantage of the [KEDA scaler for Event Hubs](#). Event-based scaling currently isn't possible when the function app is hosted in a Dedicated (App Service) plan, which requires you to determine the right number of instances based on your workload.

To learn more, see [Azure Event Hubs bindings for Azure Functions](#) and [Azure Event Hubs trigger for Azure Functions](#).

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [David Barkol](#) | Principal Solution Specialist GBB

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

[Performance and scale](#)

Related resources

- [Monitoring serverless event processing](#) provides guidance on monitoring serverless event-driven architectures.
- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the reference architecture work.

Performance and scale guidance for Event Hubs and Azure Functions

Azure Event Hubs

Azure Functions

This article provides guidance for optimizing scalability and performance when you use Azure Event Hubs and Azure Functions together in your applications.

Function grouping

Typically, a function encapsulates a unit of work in an event-processing stream. For instance, a function can transform an event into a new data structure or enrich data for downstream applications.

In Functions, a function app provides the execution context for functions. Function app behaviors apply to all functions that the function app hosts. Functions in a function app are deployed together and scaled together. All functions in a function app must be of the same language.

How you group functions into function apps can affect the performance and scaling capabilities of your function apps. You can group according to access rights, deployment, and the usage patterns that invoke your code.

For guidance on Functions best practices for grouping and other aspects, see [Best practices for reliable Azure Functions](#) and [Improve the performance and reliability of Azure Functions](#).

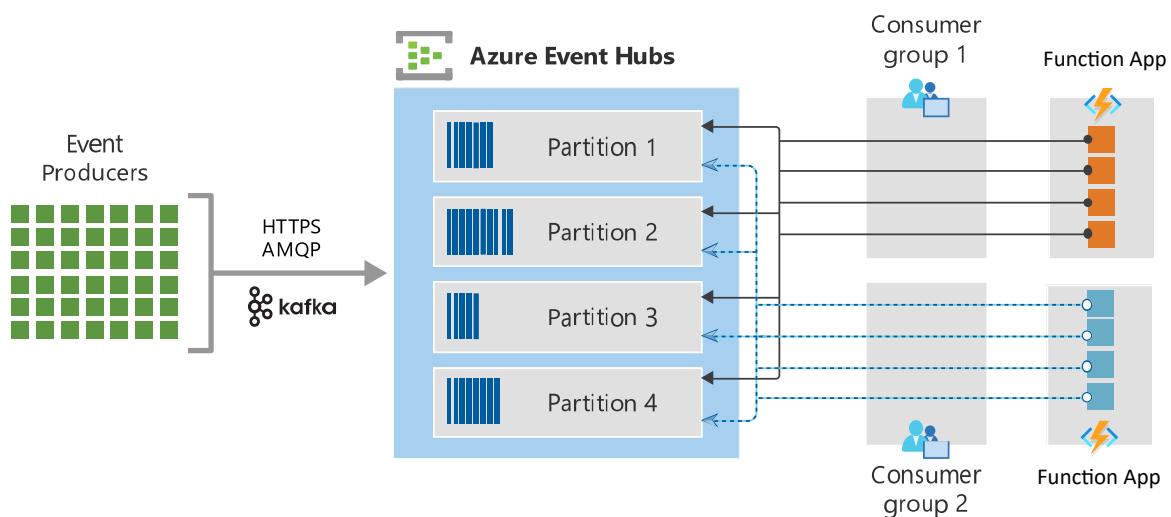
The following list is guidance for grouping functions. The guidance considers storage and consumer group aspects:

- **Host a single function in a function app:** If Event Hubs triggers a function, you can, to reduce contention between that function and other functions, isolate the function in its own function app. Isolation is especially important if the other functions are CPU or memory intensive. This technique helps because each function has its own memory footprint and usage patterns that can directly affect the scaling of the function app that hosts it.
- **Give each function app its own storage account:** Avoid sharing storage accounts between function apps. Also, if a function app uses a storage account, don't use

that account for other storage operations or needs. It can be especially important to avoid sharing storage accounts for functions that Event Hubs triggers, because such functions can have a high volume of storage transactions due to checkpointing.

- **Create a dedicated consumer group for each function app:** A consumer group is a view of an event hub. Different consumer groups have different views, which means that the states, positions, and offsets can differ. Consumer groups make it possible for multiple consuming applications to have their own views of the event stream, and to read the stream independently at their own pace and with their own offsets. For more information about consumer groups, see [Features and terminology in Azure Event Hubs](#).

A consumer group has one or more consumer applications associated with it, and a consumer application can use one or more consumer groups. In a stream processing solution, each consumer application equates to a consumer group. A function app is a prime example of a consumer application. The following diagram provides an example of two function apps that read from an event hub, where each app has its own dedicated consumer group:



Don't share consumer groups between function apps and other consumer applications. Each function app should be a distinct application with its own assigned consumer group to ensure offset integrity for each consumer and to simplify dependencies in an event streaming architecture. Such a configuration, along with providing each event hub-triggered function its own function app and storage account, helps set the foundation for optimal performance and scaling.

Function hosting plans

Every function app is hosted according to one of three hosting plans. For information about these plans, see [Azure Functions hosting options](#). Take note of how the three options scale.

The Consumption plan is the default. Function apps in the Consumption plan scale independently and are most effective when they avoid long-running tasks.

The Premium and Dedicated plans are often used to host multiple function apps and functions that are more CPU and memory intensive. With the Dedicated plan, you run your functions in an Azure App Service plan at regular App Service plan rates. It's important to note that all the function apps in these plans share the resources that are allocated to the plan. If functions have different load profiles or unique requirements, it's best to host them in different plans, especially in stream processing applications.

Event Hubs scaling

When you deploy an Event Hubs namespace, there are several important settings that you need to set properly to ensure peak performance and scaling. This section focuses on the Standard tier of Event Hubs and the unique features of that tier that affect scaling when you also use Functions. For more information about Event Hubs tiers, see [Basic vs. Standard vs. Premium vs. Dedicated tiers](#).

An Event Hubs namespace corresponds to a Kafka cluster. For information about how Event Hubs and Kafka relate to one another, see [What is Azure Event Hubs for Apache Kafka](#).

Understanding throughput units (TUs)

In the Event Hubs Standard tier, throughput is classified as the amount of data that enters and is read from the namespace per unit of time. TUs are pre-purchased units of throughput capacity.

TUs are billed on an hourly basis.

All the event hubs in a namespace share the TUs. To properly calculate capacity needs, you must consider all the applications and services, both publishers and consumers. Functions affect the number of bytes and events that are published to and read from an event hub.

The emphasis for determining the number of TUs is on the point of ingress. However, the aggregate for the consumer applications, including the rate at which those events are processed, must also be included in the calculation.

For more information Event Hubs throughput units, see [Throughput units](#).

Scale up with Auto-inflate

Auto-inflate can be enabled on an Event Hubs namespace to accommodate situations in which the load exceeds the configured number of TUs. Using Auto-inflate prevents throttling of your application and helps ensure that processing, including the ingesting of events, continues without disruption. Because the TU setting affects costs, using Auto-inflate helps address concerns about overprovisioning.

Auto-inflate is a feature of Event Hubs that's often confused with autoscale, especially in the context of serverless solutions. However, Auto-inflate, unlike autoscale, doesn't scale-down when added capacity is no longer needed.

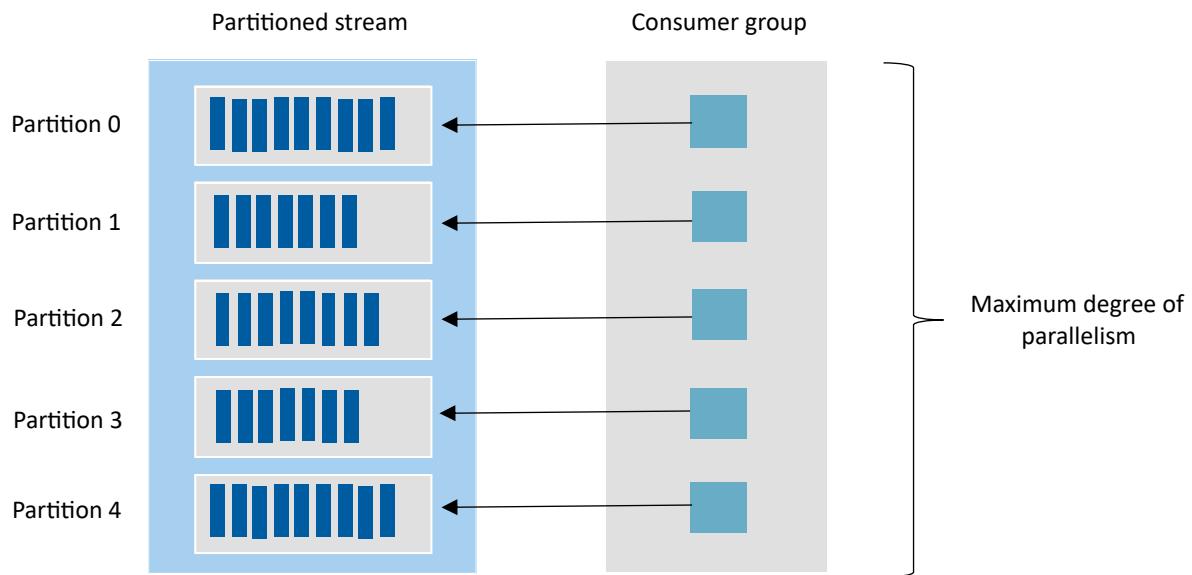
If the application needs capacity that exceeds the maximum allowed number of TUs, consider using Event Hubs [Premium tier](#) or [Dedicated tier](#).

Partitions and concurrent functions

When an event hub is created, the number of [partitions](#) must be specified. The partition count remains fixed and can't be changed except from the Premium and Dedicated tiers. When Event Hubs triggers functions apps, it's possible that the number of concurrent instances can equal the number of partitions.

In Consumption and Premium hosting plans, the function app instances scale out dynamically to the meet the number of partitions, if needed. The Dedicated hosting plan runs functions in an App Service plan and requires that you manually configure your instances or set up an autoscale scheme. For more information, see [Dedicated hosting plans for Azure Functions](#).

Ultimately, a one-to-one relationship between the number of partitions and function app instances is the ideal target for maximum throughput in a stream processing solution. To achieve optimal parallelism, have multiple consumers in a consumer group. For Functions, this objective translates to many instances of a function app in the plan. The result is referred to as *partition-level parallelism* or the *maximum degree of parallelism*, as shown in the following diagram:



It might seem to make sense to configure as many partitions as possible to achieve maximum throughput and to account for the possibility of a higher volume of events. However, there are several important factors to consider when you configure many partitions:

- **More partitions can lead to more throughput:** Because the degree of parallelism is the number of consumers (function app instances), the more partitions there are, the higher the concurrent throughput can be. This fact is important when you share a designated number of TUs for an event hub with other consumer applications.
- **More functions can require more memory:** As the number of function app instances increases, so does the memory footprint of resources in the plan. At some point, too many partitions can deteriorate performance for consumers.
- **There's a risk of back pressure from downstream services:** As more throughput is generated, you run the risk of overwhelming downstream services or receiving back pressure from them. Consumer fan-out must be accounted for when considering the consequences to surrounding resources. Possible consequences included throttling from other services, network saturation, and other forms of resource contention.
- **Partitions can be sparsely populated:** The combination of many partitions and a low volume of events can lead to data that's sparsely distributed across partitions. Instead, a smaller number of partitions can provide better performance and resource usage

Availability and consistency

When a partition key or ID isn't specified, Event Hubs routes an incoming event to the next available partition. This practice provides high availability and helps increase throughput for consumers.

When ordering of a set of events is required, the event producer can specify that a particular partition is to be used for all the events of the set. The consumer application that reads from the partition receives the events in proper order. This tradeoff provides consistency but compromises availability. Don't use this approach unless the order of events must be preserved.

For Functions, ordering is achieved when events are published to a particular partition and an Event Hubs triggered function obtains a lease to the same partition. Currently, the ability to configure a partition with the Event Hubs output binding isn't supported. Instead, the best approach is to use one of the Event Hubs SDKs to publish to a specific partition.

For more information about how Event Hubs supports availability and consistency, see [Availability and consistency in Event Hubs](#).

Event Hubs trigger

This section focuses on the settings and considerations for optimizing functions that Event Hubs triggers. Factors include batch processing, sampling, and related features that influence the behavior of an event hub trigger binding.

Batching for triggered functions

You can configure functions that an event hub triggers to process a batch of events or one event at a time. Processing a batch of events is more efficient because it eliminates some of the overhead of function invocations. Unless you need to process only a single event, your function should be configured to process multiple events when invoked.

Enabling batching for the Event Hubs trigger binding varies between languages:

- JavaScript, Python, and other languages enable batching when the **cardinality** property is set to **many** in the `function.json` file for the function.
- In C#, **cardinality** is automatically configured when an array is designated for the type in the `EventHubTrigger` attribute.

For more information about how batching is enabled, see [Azure Event Hubs trigger for Azure Functions](#).

Trigger settings

Several configuration settings in the `host.json` file play a key role in the performance characteristics of the Event Hubs trigger binding for Functions:

- **maxEventBatchSize:** This setting represents the maximum number of events that the function can receive when it's invoked. If the number of events received is less than this amount, the function is still invoked with as many events as are available. You can't set a minimum batch size.
- **prefetchCount:** The prefetch count is one of the most important settings when you optimize for performance. The underlying AMQP channel references this value to determine how many messages to fetch and cache for the client. The prefetch count should be greater than or equal to the **maxEventBatchSize** value and is commonly set to a multiple of that amount. Setting this value to a number less than the **maxEventBatchSize** setting can hurt performance.
- **batchCheckpointFrequency:** As your function processes batches, this value determines the rate at which checkpoints are created. The default value is 1, which means that there's a checkpoint whenever a function successfully processes a batch. A checkpoint is created at the partition level for each reader in the consumer group. For information about how this setting influences replays and retries of events, see [Event hub triggered Azure function: Replays and Retries \(blog post\)](#) ↗.

Do several performance tests to determine the values to set for the trigger binding. We recommend that you change settings incrementally and measure consistently to fine-tune these options. The default values are a reasonable starting point for most event processing solutions.

Checkpointing

Checkpoints mark or commit reader positions in a partition event sequence. It's the responsibility of the Functions host to checkpoint as events are processed and the setting for the batch checkpoint frequency is met. For more information about checkpointing, see [Features and terminology in Azure Event Hubs](#).

The following concepts can help you understand the relationship between checkpointing and the way that your function processes events:

- **Exceptions still count towards success:** If the function process doesn't crash while processing events, the completion of the function is considered successful, even if exceptions occurred. When the function completes, the Functions host evaluates **batchCheckpointFrequency**. If it's time for a checkpoint, it creates one, regardless of whether there were exceptions. The fact that exceptions don't affect checkpointing shouldn't affect your proper use of exception checking and handling.
- **Batch frequency matters:** In high-volume event streaming solutions, it can be beneficial to change the **batchCheckpointFrequency** setting to a value greater

than 1. Increasing this value can reduce the rate of checkpoint creation and, as a consequence, the number of storage I/O operations.

- **Replays can happen:** Each time a function is invoked with the Event Hubs trigger binding, it uses the most recent checkpoint to determine where to resume processing. The offset for every consumer is saved at the partition level for each consumer group. Replays happen when a checkpoint doesn't occur during the last invocation of the function, and the function is invoked again. For more information about duplicates and deduplication techniques, see [Idempotency](#).

Understanding checkpointing becomes critical when you consider best practices for error handling and retries, a topic that's discussed later in this article.

Telemetry sampling

Functions provides built-in support for Application Insights, an extension of Azure Monitor that provides application performance monitoring capabilities. With this feature, you can log information about function activities, performance, runtime exceptions, and more. For more information, see [Application Insights overview](#).

This powerful capability offers some key configuration choices that affect performance. Some of the notable settings and considerations for monitoring and performance are:

- **Enable telemetry sampling:** For high-throughput scenarios, you should evaluate the amount of telemetry and information that you need. Consider using the [telemetry sampling](#) feature in Application Insights to avoid degrading the performance of your function with unnecessary telemetry and metrics.
- **Configure aggregation settings:** Examine and configure the frequency of aggregating and sending data to Application Insights. This configuration setting is in the [host.json](#) file along with many other sampling and logging related options. For more information, see [Configure the aggregator](#).
- **Disable AzureWebJobDashboard:** For apps that target version 1.x of the Functions runtime, this setting stores the connection string to a storage account that the Azure SDK uses to retain logs for the WebJobs dashboard. If Application Insights is used instead of the WebJobs dashboard, then this setting should be removed. For more information, see [AzureWebJobsDashboard](#).

When Application Insights is enabled without sampling, all telemetry is sent. Sending data about all events can have a detrimental effect on the performance of the function, especially under high-throughput event streaming scenarios.

Taking advantage of sampling and continually assessing the appropriate amount of telemetry needed for monitoring is crucial for optimum performance. Telemetry should

be used for general platform health evaluation and for occasional troubleshooting, not to capture core business metrics. For more information, see [Configure sampling](#).

Output binding

Use the [Event Hubs output binding for Azure Functions](#) to simplify publishing to an event stream from a function. The benefits of using this binding include:

- **Resource management:** The binding handles both the client and connection lifecycles for you, and reduces the potential for issues that can arise with port exhaustion and connection pool management.
- **Less code:** The binding abstracts the underlying SDK and reduces the amount of code that you need to publish events. It helps you write code that's easier to write and maintain.
- **Batching:** For several languages, batching is supported to efficiently publish to an event stream. Batching can improve performance and help streamline the code that sends the events.

We strongly recommend that you review the list of [Languages that Functions supports](#) and the developer guides for those languages. The **Bindings** section for each language provides detailed examples and documentation.

Batching when publishing events

If your function only publishes a single event, configuring the binding to return a value is a common approach that's helpful if the function execution always ends with a statement that sends the event. This technique should only be used for synchronous functions that return only one event.

Batching is encouraged to improve performance when sending multiple events to a stream. Batching allows the binding to publish events in the most efficient possible way.

Support for using the output binding to send multiple events to Event Hubs is available in C#, Java, Python, and JavaScript.

Output multiple events (C#)

Use the **ICollector** and **IAsyncCollector** types when you send multiple events from a function in C#.

- The **ICollector<T>.Add()** method can be used in both synchronous and asynchronous functions. It executes the add operation as soon as it's called.

- The `IAsyncCollector<T>.AddAsync()` method prepares the events to be published to the event stream. If you write an asynchronous function, you should use `IAsyncCollector` to better manage the published events.

For examples of using C# to publish single and multiple events, see [Azure Event Hubs output binding for Azure Functions](#).

Throttling and back pressure

Throttling considerations apply to output binding, not only for Event Hubs but also for Azure services such as [Azure Cosmos DB](#). It's important to become familiar with the limits and quotas that apply to those services and to plan accordingly.

To handle downstream errors, you can wrap `AddAsync` and `FlushAsync` in an exception handler for .NET Functions in order to catch exceptions from `IAsyncCollector`. Another option is to use the Event Hubs SDKs directly instead of using output bindings.

Function code

This section covers the key areas that must be considered when writing code to process events in a function that Event Hubs triggers.

Asynchronous programming

We recommend that you write your function to [use async code and avoid blocking calls](#), especially when I/O calls are involved.

Here are guidelines that you should follow when you write a function to process asynchronously:

- **All asynchronous or all synchronous:** If a function is configured to run asynchronously, all the I/O calls should be asynchronous. In most cases, partially asynchronous code is worse than code that's entirely synchronous. Choose either asynchronous or synchronous, and stick with the choice all the way through.
- **Avoid blocking calls:** Blocking calls return to the caller only after the call completes, in contrast to asynchronous calls that return immediately. An example in C# would be calling `Task.Result` or `Task.Wait` on an asynchronous operation.

More about blocking calls

Using blocking calls for asynchronous operations can lead to thread-pool starvation and cause the function process to crash. The crash happens because a blocking call requires another thread to be created to compensate for the original call that's now waiting. As a result, it requires twice as many threads to complete the operation.

Avoiding this *sync over async* approach is especially important when Event Hubs is involved, because a function crash doesn't update the checkpoint. The next time the function is invoked it could end up in this cycle and appear to be stuck or to move along slowly as function executions eventually time out.

Troubleshooting this phenomenon usually starts with reviewing the trigger settings and running experiments that can involve increasing the partition count. Investigations can also lead to changing several of the batching options such as the max batch size or prefetch count. The impression is that it's a throughput problem or configuration setting that just needs to be tuned accordingly. However, the core problem is in the code itself and must be addressed there.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributor.

Principal author:

- [David Barkol](#) | Principal Solution Specialist GBB

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Before continuing, consider reviewing these related articles:

- [Monitor executions in Azure Functions](#)
- [Azure Functions reliable event processing](#)
- [Designing Azure Functions for identical input](#)
- [ASP.NET Core async guidance](#).
- [Azure Event Hubs trigger for Azure Functions](#)

[Resilient Event Hubs and Functions design](#)

Related resources

- Monitoring serverless event processing
- Serverless event processing
- De-batching and filtering in serverless event processing with Event Hubs

Resilient Event Hubs and Functions design

Article • 10/28/2022

Error handling, designing for idempotency and managing retry behavior are a few of the critical measures you can take to ensure Event Hubs triggered functions are resilient and capable of handling large volumes of data. This article covers these crucial concepts and makes recommendations for serverless event-streaming solutions.

Azure provides three main messaging services that can be used with Azure Functions to support a wide range of unique, event-driven scenarios. Because of its partitioned consumer model and ability to ingest data at a high rate, Azure Event Hubs is commonly used for event streaming and big data scenarios. For a detailed comparison of Azure messaging services, see [Choose between Azure messaging services - Event Grid, Event Hubs, and Service Bus](#).

Streaming benefits and challenges

Understanding the benefits and drawbacks of streams helps you appreciate how a service like [Event Hubs](#) operates. You also need this context when making impactful architectural decisions, troubleshooting issues, and optimizing for performance.

Consider the following key concepts about solutions featuring both Event Hubs and Functions:

- **Streams are not queues:** Event Hubs, Kafka, and other similar offerings that are built on the partitioned consumer model don't intrinsically support some of the principal features in a message broker like [Service Bus](#). Perhaps the biggest indicator of this is the fact that reads are **non-destructive**. This means that the data that is read by the Functions host isn't deleted afterwards. Instead, messages are immutable and remain for other consumers to read, including potentially the same customer reading it again. For this reason, solutions that implement patterns such as [competing consumers](#) are better suited for a traditional message broker.
- **Missing inherent dead-letter support:** A dead-letter channel is not a native feature in Event Hubs or Kafka. Often, the *concept* of dead-lettering is integrated into a streaming solution to account for data that cannot be processed. This functionality is intentionally not an innate element in Event Hubs and is only added on the consumer side to manufacture a similar behavior or effect. If you need dead-letter support, you should potentially review your choice of streaming message service.

- **A unit of work is a partition:** In a traditional message broker, a unit of work is a single message. In a streaming solution, a partition is often considered the unit of work. If each event in an event hub is regarded as a discrete message that requires it to be treated like an order processing operation or financial transaction, it's most likely an indication of the wrong messaging service being used.
- **No server-side filtering:** One of the reasons Event Hubs is capable of tremendous scale and throughput is due to the low overhead on the service itself. Features like server-side filtering, indexes, and cross-broker coordination aren't part of the architecture of Event Hubs. Functions are occasionally used to filter events by routing them to other Event Hubs based on the contents in the body or header. This approach is common in event streaming but comes with the caveat that each event is read and evaluated by the initial function.
- **Every reader must read all data:** Since server-side filtering is unavailable, a consumer sequentially reads all the data in a partition. This includes data that may not be relevant or could even be malformed. There are several options and even strategies that can be used to compensate for these challenges that will be covered later in this section.

These significant design decisions allow Event Hubs to do what it does best: support a significant influx of events and provide a robust and resilient service for consumers to read from. Each consumer application is tasked with the responsibility of maintaining their own, client-side offsets or cursor to those events. The low overhead makes Event Hubs an affordable and powerful option for event streaming.

Idempotency

One of the core tenets of Azure Event Hubs is the concept of at-least once delivery. This approach ensures that events will always be delivered. It also means that events can be received more than once, even repeatedly, by consumers such as a function. For this reason, it's important that an event hub triggered function supports the [idempotent consumer](#) pattern.

Working under the assumption of at-least once delivery, especially within the context of an event-driven architecture, is a responsible approach for reliably processing events. Your function must be idempotent so that the outcome of processing the same event multiple times is the same as processing it once.

Duplicate events

There are several different scenarios that could result in duplicate events being delivered to a function:

- **Checkpointing:** If the Azure Functions host crashes, or the threshold set for the [batch checkpoint frequency](#) is not met, a checkpoint will not be created. As a result, the offset for the consumer is not advanced and the next time the function is invoked, it will resume from the last checkpoint. It is important to note that checkpointing occurs at the partition level for each consumer.
- **Duplicate events published:** There are many techniques that could alleviate the possibility of the same event being published to a stream, however, it's still the responsibility of the consumer to idempotently handle duplicates.
- **Missing acknowledgments:** In some situations, an outgoing request to a service may be successful, however, an acknowledgment (ACK) from the service is never received. This might result in the perception that the outgoing call failed and initiate a series of retries or other outcomes from the function. In the end, duplicate events could be published, or a checkpoint is not created.

Deduplication techniques

Designing your functions for [identical input](#) should be the default approach taken when paired with the Event Hub trigger binding. You should consider the following techniques:

- **Looking for duplicates:** Before processing, take the necessary steps to validate that the event should be processed. In some cases, this will require an investigation to confirm that it is still valid. It could also be possible that handling the event is no longer necessary due to data freshness or logic that invalidates the event.
- **Design events for idempotency:** By providing additional information within the payload of the event, it may be possible to ensure that processing it multiple times will not have any detrimental effects. Take the example of an event that includes an amount to withdrawal from a bank account. If not handled responsibly, it is possible that it could decrement the balance of an account multiple times. However, if the same event includes the updated balance to the account, it could be used to perform an upsert operation to the bank account balance. This event-carried state transfer approach occasionally requires coordination between producers and consumers and should be used when it makes sense to participating services.

Error handling and retries

Error handling and retries are a few of the most important qualities of distributed, event-driven applications, and Functions are no exception. For event streaming solutions, the need for proper error handling support is crucial, as thousands of events can quickly turn into an equal number of errors if they are not handled correctly.

Error handling guidance

Without error handling, it can be tricky to implement retries, detect runtime exceptions, and investigate issues. Every function should have at least some level of error handling. A few recommended guidelines are:

- **Use Application Insights:** Enable and use Application Insights to log errors and monitor the health of your functions. Be mindful of the configurable sampling options for scenarios that process a high volume of events.
- **Add structured error handling:** Apply the appropriate error handling constructs for each programming language to catch, log, and detect anticipated and unhandled exceptions in your function code. For instance, use a try/catch block in C#, Java and JavaScript and take advantage of the [try and except](#) blocks in Python to handle exceptions.
- **Logging:** Catching an exception during execution provides an opportunity to log critical information that could be used to detect, reproduce, and fix issues reliably. Log the exception, not just the message, but the body, inner exception and other useful artifacts that will be helpful later.
- **Do not catch and ignore exceptions:** One of the worst things you can do is catch an exception and do nothing with it. If you catch a generic exception, log it somewhere. If you don't log errors, it's difficult to investigate bugs and reported issues.

Retries

Implementing retry logic in an event streaming architecture can be complex. Supporting cancellation tokens, retry counts and exponential back off strategies are just a few of the considerations that make it challenging. Fortunately, Functions provides [retry policies](#) that can make up for many of these tasks that you would typically code yourself.

Several important factors that must be considered when using the retry policies with the Event Hub binding, include:

- **Avoid indefinite retries:** When the [max retry count](#) setting is set to a value of -1, the function will retry indefinitely. In general, indefinite retries should be used

sparingly with Functions and almost never with the Event Hub trigger binding.

- **Choose the appropriate retry strategy:** A [fixed delay](#) strategy may be optimal for scenarios that receive back pressure from other Azure services. In these cases, the delay can help avoid throttling and other limitations encountered from those services. The [exponential back off](#) strategy offers more flexibility for retry delay intervals and is commonly used when integrating with third-party services, REST endpoints, and other Azure services.
- **Keep intervals and retry counts low:** When possible, try to maintain a retry interval shorter than one minute. Also, keep the maximum number of retry attempts to a reasonably low number. These settings are especially pertinent when running in the Functions Consumption plan.
- **Circuit breaker pattern:** A transient fault error from time to time is expected and a natural use case for retries. However, if a significant number of failures or issues are occurring during the processing of the function, it may make sense to stop the function, address the issues and restart later.

An important takeaway for the retry policies in Functions is that it is a best effort feature for reprocessing events. It does not substitute the need for error handling, logging, and other important patterns that provide resiliency to your code.

Strategies for failures and corrupt data

There are several noteworthy approaches that you can use to compensate for issues that arise due to failures or bad data in an event stream. Some fundamental strategies are:

- **Stop sending and reading:** Pause the reading and writing of events to fix the underlying issue. The benefit of this approach is that data won't be lost, and operations can resume after a fix is rolled out. This approach may require a circuit-breaker component in the architecture and possibly a notification to the affected services to achieve a pause. In some cases, stopping a function may be necessary until the issues are resolved.
- **Drop messages:** If messages aren't important or are considered non-mission critical, consider moving on and not processing them. This doesn't work for scenarios that require strong consistency such as recording moves in a chess match or finance-based transactions. Error handling inside of a function is recommended for catching and dropping messages that can't be processed.
- **Retry:** There are many situations that may warrant the reprocessing of an event. The most common scenario would be a transient error encountered when calling

another service or dependency. Network errors, service limits and availability, and strong consistency are perhaps the most frequent use cases that justify reprocessing attempts.

- **Dead letter:** The idea here is to publish the event to a different event hub so that the existing flow is not interrupted. The perception is that it has been moved off the hot path and could be dealt with later or by a different process. This solution is used frequently for handling poisoned messages or events. It should be noted that each function, that is configured with a different consumer group, will still encounter the bad or corrupt data in their stream and must handle it responsibly.
- **Retry and dead letter:** The combination of numerous retry attempts before ultimately publishing to a dead letter stream once a threshold is met, is another familiar method.
- **Use a schema registry:** A schema registry can be used as a proactive tool to help improve consistency and data quality. The [Azure Schema Registry](#) can support the transition of schemas along with versioning and different compatibility modes as schemas evolve. At its core, the schema will serve as a contract between producers and consumers, which could reduce the possibility of invalid or corrupt data being published to the stream.

In the end, there isn't a perfect solution and the consequences and tradeoffs of each of the strategies needs to be thoroughly examined. Based on the requirements, using several of these techniques together may be the best approach.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [David Barkol](#) | Principal Solution Specialist GBB

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Before continuing, consider reviewing these related articles:

- [Azure Functions reliable event processing](#)
- [Designing Azure Functions for identical input](#)

- Azure Functions error handling and retry guidance

Security

Related resources

- [Monitoring serverless event processing](#) provides guidance on monitoring serverless event-driven architectures.
- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the reference architecture work.

Secure Azure Functions with Event Hubs

Article • 10/10/2023

When configuring access to resources in Azure, you should apply fine-grained control over permissions to resources. Access to these resources should be based on *need to know* and *least privilege* security principles to make sure that clients can only perform the limited set of actions assigned to them.

Authorizing Access to Event Hubs

Authorizing access to Azure Event Hubs resources can be done using the following security constructs:

- **Microsoft Entra ID:** Microsoft Entra ID provides role-based access control (RBAC) for granular control over a client's access to Event Hubs resources. Based on roles and permissions granted, Microsoft Entra ID will authorize requests using an OAuth 2.0 access token.
- **Shared access signature:** A shared access signature (SAS) offers the ability to protect Event Hubs resources based on authorization rules. You define authorization policies by selecting one or more [policy rules](#), such as the ability to send messages, listen to messages, and manage the entities in the namespace.

Shared access signature considerations

When using a shared access signature with Azure Functions and Event Hubs, the following considerations should be reviewed:

- **Avoid the Manage right:** In addition to being able to manage the entities in an Event Hubs namespace, the Manage right includes both Send and Listen rights. Ideally, a function app should only be granted a combination of the Send and Listen rights, based on the actions they perform.
- **Don't use the default Manage rule:** Avoid using the default policy rule named `RootManageSharedAccessKey` unless it's needed by your function app, which should be an uncommon scenario. Another caveat to this default rule is that it's created at the namespace level and grants permissions to all underlying event hubs.
- **Review shared access policy scopes:** Shared access policies can be created at the namespace level and per event hub. Consider creating granular access policies that

are tailored for each client to limit their range and permissions.

Managed identity

An Active Directory identity can be assigned to a managed resource in Azure such as a function app or web app. Once an identity is assigned, it has the capabilities to work with other resources that use Microsoft Entra ID for authorization, much like a [service principal](#).

Function apps can be assigned a [managed identity](#) and take advantage of identity-based connections for a subset of services, including Event Hubs. Identity-based connections provide support for both the trigger and output binding extensions and must use the [Event Hubs extension 5.x and higher](#) for support.

Network

By default, Event Hubs namespaces are accessible from the internet, so long as the request comes with valid authentication and authorization. There are three options for limiting network access to Event Hubs namespaces:

- [Allow access from specific IP addresses](#)
- [Allow access from specific virtual networks \(service endpoints\)](#)
- [Allow access via private endpoints](#)

In all cases, it's important to note that at least one IP firewall rule or virtual network rule for the namespace is specified. Otherwise, if no IP address or virtual network rule is specified, the namespace is accessible over the public internet (using the access key).

Azure Functions can be configured to consume events from or publish events to event hubs, which are set up with either service endpoints or private endpoints. Regional virtual network integration is needed for your function app to connect to an event hub using a service endpoint or a private endpoint.

When setting up Functions to work with a private endpoint enabled resource, you need to set the `WEBSITE_VNET_ROUTE_ALL` application setting to `1`. If you want to fully lock down your function app, you also need to [restrict your storage account](#).

To trigger (consume) events in a virtual network environment, the function app needs to be hosted in a Premium plan, a Dedicated (App Service) plan, or an App Service Environment (ASE).

Additionally, running in an Azure Functions Premium plan and consuming events from a virtual network restricted Event Hub requires virtual network trigger support, also

referred to as [runtime scale monitoring](#). Runtime scale monitoring can be configured via the Azure portal, Azure CLI, or other deployment solutions. Runtime scale monitoring isn't available when the function is running in a Dedicated (App Service) plan or an ASE.

To use runtime scale monitoring with Event Hubs, you need to use version 4.1.0 or higher of the Microsoft.Azure.WebJobs.Extensions.EventHubs extension.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [David Barkol](#) | Principal Solution Specialist GBB

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Before continuing, consider reviewing these related articles:

- [Authorize access with Microsoft Entra ID](#)
- [Authorize access with a shared access signature in Azure Event Hubs](#)
- [Configure an identity-based resource](#)

Observability

Related resources

- [Monitoring serverless event processing](#) provides guidance on monitoring serverless event-driven architectures.
- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the reference architecture work.

Monitor Azure Functions and Event Hubs

Azure Event Hubs

Azure Functions

Azure Monitor

Monitoring provides insights into the behavior and health of your systems, and helps build a holistic view of the environment, historic trends, correlate diverse factors, and measure changes in performance, consumption, or error rate.

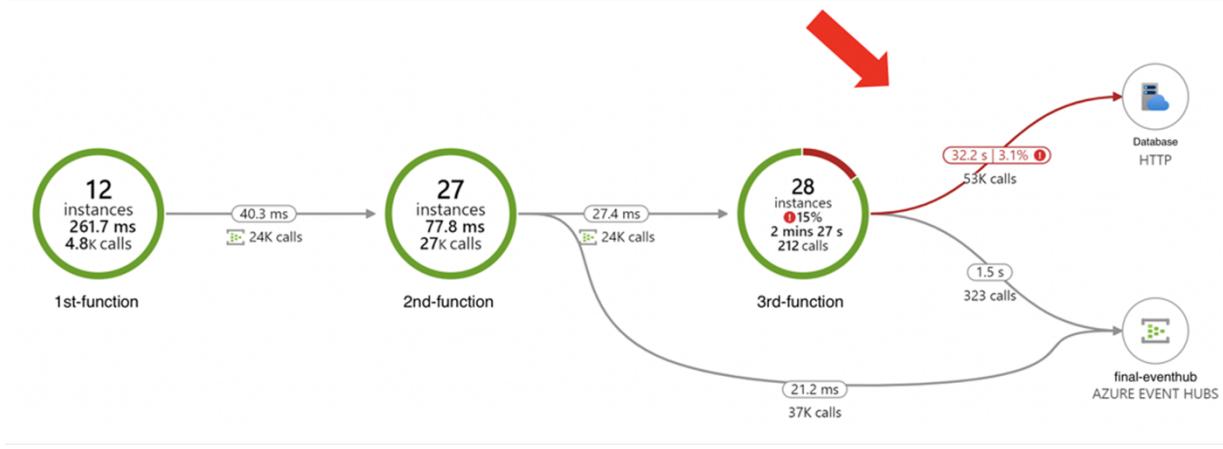
Azure Functions offers built-in integration with [Application Insights](#). From Application Insights, you can get information such as the number of function app instances or request and dependency telemetry of a function. When working with Functions and Event Hubs, Application Insights can also track the outgoing dependency telemetries to the event hub, calculating the time spent in the queue and showing the end-to-end flow of the system connected through Event Hubs.

This section introduces useful features and insights that you can get from Application Insights for your Event Hubs plus Functions solution.

Application Map

[Application Map](#) shows how the components in a system are interacting with each other. Because of the dependency telemetry that Application Insights provides, it maps out the flow of events between Azure Functions and Event Hubs, including the average of each function execution and average duration of an event in Event Hubs, as well as showing transactions that contain failures marked in red.

After sending the expected load to your system, you can go to Application Insights in the [Azure portal](#) ↗, and on the sidebar, choose on **Application Map**. Here's a map that shows three functions, three event hubs, and apparent failures when writing to a downstream database:



End-to-end transaction details

End-to-end transaction details show how your system components interact with each other, in chronological order. This view also shows how long an event has spent in the queue. You can also drill into the telemetry of each component from this view, which makes it easier to troubleshoot across components within the same request when an issue occurred.

Home > Application Insights > fordemofirstfunction > Performance >

End-to-end transaction details ↗

fordemofirstfunction

All Sort by Relevance

11/6/2020, 11:19:59 AM FirstFunction →

Duration: 19.9 ms Response code: 0

11/5/2020, 3:09:41 PM FirstFunction →

Duration: 30.1 ms Response code: 0

11/5/2020, 3:03:27 PM FirstFunction →

Duration: 24.4 ms Response code: 0

11/5/2020, 3:28:42 PM FirstFunction →

Duration: 16.9 ms Response code: 0

11/5/2020, 3:23:20 PM FirstFunction →

Search results Learn more Copy link Feedback Leave preview

End-to-end transaction
Operation ID: 2295ea6e8de7aa4f87c49e711b60a1fc, d7d48c7d1af16c4eb129fe517a87608b ↗

EVENT 0 10 MS 20 MS 30 MS

fordemofirstfunction FirstFunction fordemosecondfunction SecondFunction

testevh Send Time Spent in Queue fordemosecondfunction SecondFunction

Platform metrics and telemetry

Platform-generated metrics in Azure Monitor for Event Hubs and Azure Functions can be used for overall monitoring of the solution behavior and health:

- [Azure Event Hubs metrics in Azure Monitor](#) are of interest to capture useful insights for Event Hubs (like aggregates of Incoming Requests, Outgoing Requests, Throttled Requests, Successful Requests, Incoming Messages, Outgoing Messages, Captured Messages, Incoming Bytes, Outgoing Bytes, Captured Bytes, User Errors).

- Azure Functions metrics share many of the metrics from [Azure App Service](#), with the addition of [Function Execution Count](#) and [Function Execution Units](#) that can be used for [understanding utilization and cost of the Consumption plan](#). Other metrics of interest are Connections, Data In, Data Out, Average Memory Working Set, Thread Count, Requests, and Response Time.

Azure Functions integrates with Application Insights to provide advanced and detailed telemetry and insights into the Functions host and function executions. To learn more, see [Analyze Azure Functions telemetry in Application Insights](#). When using Application Insights to monitor a topology, there are a variety of configurations available. To learn more, see [How to configure monitoring for Azure Functions](#).

The following is an example of extra telemetry for Event Hubs triggered functions generated in the **traces** table:

```
Trigger Details: PartitionId: 6, Offset: 3985758552064-3985758624640,  
EnqueueTimeUtc: 2022-10-31T12:51:58.1750000+00:00-2022-10-  
31T12:52:03.8160000+00:00, SequenceNumber: 3712266-3712275, Count: 10
```

This information requires that you use Event Hubs extension 4.2.0 or a later version. This data is very useful as it contains information about the message that triggered the function execution and can be used for querying and insights. It includes the following data for each time the function is triggered:

- The **partition ID** (6)
- The **partition offset** range (3985758552064-3985758624640)
- The **Enqueue Time range** in UTC (2022-10-31T12:51:58.1750000+00:00-2022-10-31T12:52:03.8160000+00:00)
- The **sequence number range** 3712266-3712275
- And the **count of messages** (10)

Refer to the [Example Application Insights queries](#) section for examples on how to use this telemetry.

Custom telemetry is also possible for different languages ([C# class library](#), [C# Isolated](#), [C# Script](#), [JavaScript](#), [Java](#), [PowerShell](#), and [Python](#)). This logging shows up in the **traces** table in Application Insights. You can create your own entries into Application Insights and add custom dimensions that can be used for querying data and creating custom dashboards.

Finally, when your function app connects to an event hub using an output binding, entries are also written to the [Application Insights Dependencies table](#).

1 dependencies

Results Chart | Columns Group columns

Completed. Showing results from the last 24 hours.

00:01.1 4,170 records

timestamp [UTC]	id	target	type	name	success	duration	performanceBucket
9/9/2021, 10:48:36.032 PM	f281e7b3f54e9...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	110.867	<250ms
9/9/2021, 10:48:36.032 PM	f281e7b3f54e9...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	110.867	<250ms
9/9/2021, 10:48:36.154 PM	030fbfe4cf55...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	18.28	<250ms
9/9/2021, 10:48:36.154 PM	030fbfe4cf55...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	18.28	<250ms
9/9/2021, 10:48:37.278 PM	b95ab2d6bdc...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	10.205	<250ms
9/9/2021, 10:48:37.278 PM	b95ab2d6bdc...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	10.205	<250ms
9/9/2021, 10:48:37.323 PM	931223fb2c633...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	20.138	<250ms
9/9/2021, 10:48:37.323 PM	931223fb2c633...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	20.138	<250ms
9/9/2021, 10:48:38.545 PM	d6f975f59d3a4...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	8.188	<250ms
9/9/2021, 10:48:38.545 PM	d6f975f59d3a4...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	8.188	<250ms
9/9/2021, 10:48:40.038 PM	089a7ad2cf8b8...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	8.935	<250ms
9/9/2021, 10:48:40.038 PM	089a7ad2cf8b8...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	8.935	<250ms
9/9/2021, 10:48:40.122 PM	141f7ae3a0dcc1...	sb://evhns-pk7qsyygv1uw-westus.servicebus.windows.net/evh...	Azure Event H...	Send	True	13.618	<250ms

For Event Hubs, the correlation is injected into the event payload, and you see a **Diagnostic-Id** property in events:

```
{
  "Body": "/your event",
  "Properties": {
    "Diagnostic-Id": "00-4d43caac0301954dbc76e5a786f6739a-6784fcb30fba654c-00",
  },
  "SystemProperties": {
    "x-opt-sequence-number": 72,
    "x-opt-offset": "4294975936",
    "x-opt-queued-time": "2020-11-04T16:46:46.955Z"
  }
}
```

This follows the [W3C Trace Context](#) format that's also used as **Operation Id** and **Operation Links** in telemetry created by Functions, which allows Application Insights to construct the correlation between event hub events and function executions, even when they're distributed.

When function processed a batch of events...

Operation Name	Telemetry type	Operation Id	Operation Links
FirstFunction	request	2295ea6e8de7aa4f87c49e711b60a1fc	
FirstFunction	trace	2295ea6e8de7aa4f87c49e711b60a1fc	
FirstFunction	dependency	2295ea6e8de7aa4f87c49e711b60a1fc	

Operation Name	Telemetry type	Operation Id	Operation Links
SecondFunction	request	d7d48c7d1af16c4eb129fe517a87608b	[{"operation_Id": "2295ea6e8de7aa4f87c49e711b60a1fc", "id": "ac0dc1c0f0a1a549"}, {"operation_Id": "2295ea6e8de7aa4f87c49e711b60a1fc", "id": "ac0dc1c0f0a1a549"}]
SecondFunction	trace	d7d48c7d1af16c4eb129fe517a87608b	

Example Application Insights queries

Below is a list of helpful Application Insights queries when monitoring Event Hubs with Azure Functions. This query display detailed information for event hub triggered function using telemetry **emitted by the Event Hubs extension 4.2.0 and greater**.

When [sampling is enabled](#) in Application Insights, there can be gaps in the data.

Detailed event processing information

The data is only emitted in the correct format when batched dispatch is used. Batch dispatch means that the function accepts multiple events for each execution, which is [recommended for performance](#). Keep in mind the following considerations:

- The `dispatchTimeMilliseconds` value approximates the length of time between when the event was written to the event hub and when it was picked up by the function app for processing.
- `dispatchTimeMilliseconds` can be negative or otherwise inaccurate because of clock drift between the event hub server and the function app.
- Event Hubs partitions are processed sequentially. A message won't be dispatched to function code for processing until all previous messages have been processed. Monitor the execution time of your functions as longer execution times will cause dispatch delays.
- The calculation uses the `enqueueTime` of the first message in the batch. Dispatch times might be lower for other messages in the batch.
- `dispatchTimeMilliseconds` is based on the point in time.

- Sequence numbers are per-partition, and duplicate processing can occur because Event Hubs does not guarantee exactly-once message delivery.

Kusto

```

traces
| where message startswith "Trigger Details: Parti"
| parse message with * "tionId: " partitionId:string , Offset: "
  offsetStart:string "-" offsetEnd:string", EnqueueTimeUtc: "
  enqueueTimeStart:datetime "+00:00-" enqueueTimeEnd:datetime "+00:00",
  SequenceNumber: "
  sequenceNumberStart:string "-" sequenceNumberEnd:string ", Count: "
  messageCount:int
| extend dispatchTimeMilliseconds = (timestamp - enqueueTimeStart) / 1ms
| project timestamp, cloud_RoleInstance, operation_Name, processId =
  customDimensions.ProcessId, partitionId, messageCount, sequenceNumberStart,
  sequenceNumberEnd, enqueueTimeStart, enqueueTimeEnd,
  dispatchTimeMilliseconds

```

Results Chart Columns Display time (UTC+00:00) Group columns

Completed. Showing results from the last 24 hours. 00:01.0 1,140 records

timestamp [UTC]	cloud_RoleInstance	operation_Name	processId	partitionId	messageCount	sequenceNumberStart
9/9/2021, 10:48:30.603 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	1	0
9/9/2021, 10:48:30.603 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	1	0
9/9/2021, 10:48:30.637 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	47	1
9/9/2021, 10:48:30.637 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	47	1
9/9/2021, 10:48:32.196 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	1	48
9/9/2021, 10:48:32.196 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	1	48
9/9/2021, 10:48:32.201 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	29	49
9/9/2021, 10:48:32.201 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	29	49
9/9/2021, 10:48:34.186 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	2	78
9/9/2021, 10:48:34.186 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	2	78
9/9/2021, 10:48:34.245 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	54	80
9/9/2021, 10:48:34.245 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	54	80
9/9/2021, 10:48:35.955 PM	351b8672336185d2f0787014e4d41799c719f218c036971b22431d...	ProcessorFunction	16592	0	1	134

Page 1 of 23 50 items per page 1 - 50 of 1140 items

Dispatch latency visualization

This query visualizes the 50th and 90th percentile event dispatch latency for a given event hub triggered function. See the above query for more details and notes.

Kusto

```

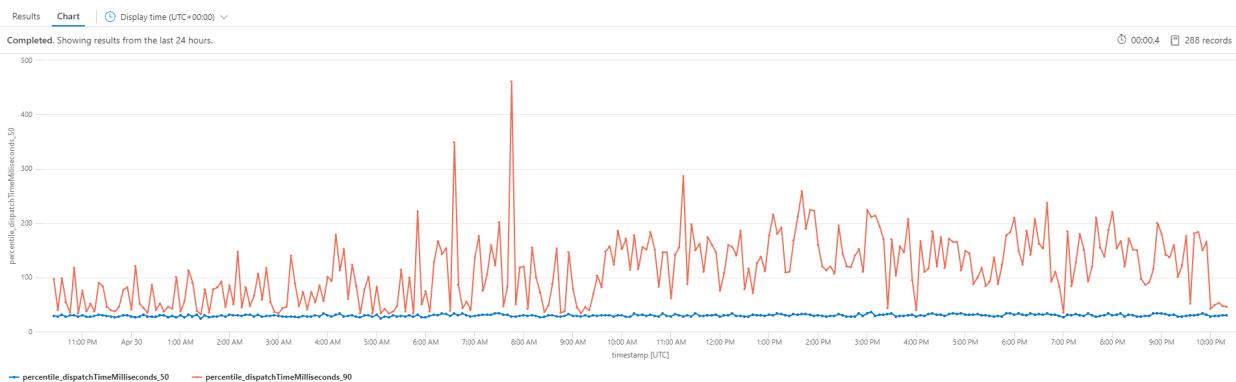
traces
| where operation_Name == "<ENTER THE NAME OF YOUR FUNCTION HERE>"
| where message startswith "Trigger Details: Parti"
| parse message with * "tionId: " partitionId:string , Offset: "
  offsetStart:string "-" offsetEnd:string", EnqueueTimeUtc: "
  enqueueTimeStart:datetime "+00:00-" enqueueTimeEnd:datetime "+00:00",
  SequenceNumber: "
  sequenceNumberStart:string "-" sequenceNumberEnd:string ", Count: "
  messageCount:int

```

```

| extend dispatchTimeMilliseconds = (timestamp - enqueueTimeStart) / 1ms
| summarize percentiles(dispatchTimeMilliseconds, 50, 90) by bin(timestamp, 5m)
| render timechart

```



Dispatch latency summary

This query is similar to above but shows a summary view.

Kusto

```

traces
| where message startswith "Trigger Details: Parti"
| parse message with * "tionId: " partitionId:string ", Offset: "
  offsetStart:string "-" offsetEnd:string", EnqueueTimeUtc: "
  enqueueTimeStart:datetime "+00:00-" enqueueTimeEnd:datetime "+00:00",
  SequenceNumber: "
  sequenceNumberStart:string "-" sequenceNumberEnd:string ", Count: "
  messageCount:int
| extend dispatchTimeMilliseconds = (timestamp - enqueueTimeStart) / 1ms
| summarize messageCount = sum(messageCount),
  percentiles(dispatchTimeMilliseconds, 50, 90, 99, 99.9, 99.99) by
  operation_Name

```

...						
Completed. Showing results from the last 24 hours.						
operation_Name	messageCount	percentile_dispatchTimeMilliseconds_50	percentile_dispatchTimeMilliseconds_90	percentile_dispatchTimeMilliseconds_99	percentile_dispatchTimeMilliseconds_99.9	percentile_dispatchTimeMilliseconds_99.99
ProcessorFunction	88,572	47.417	1,954	15,484	17.925	17.925

Message distribution across partitions

This query shows how to visualize message distribution across partitions.

Kusto

```

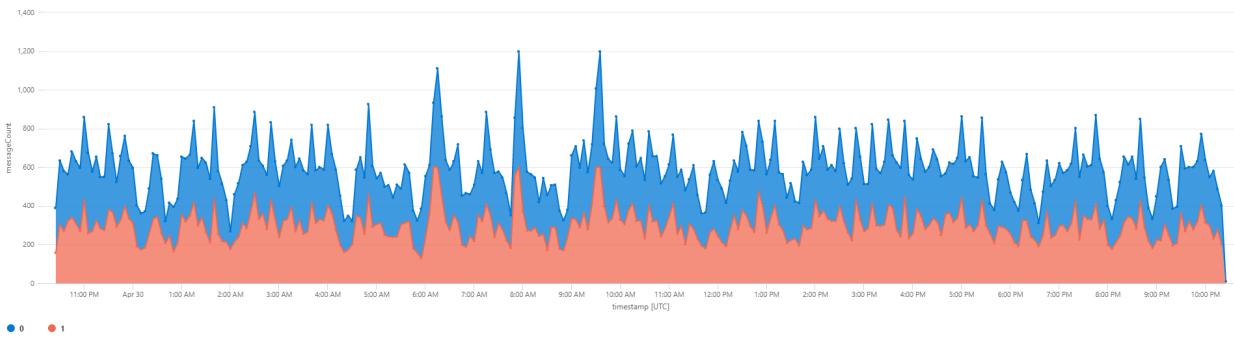
traces
| where message startswith "Trigger Details: Parti"
| parse message with * "tionId: " partitionId:string ", Offset: "

```

```

offsetStart:string "-" offsetEnd:string", EnqueueTimeUtc: "
enqueueTimeStart:datetime "+00:00-" enqueueTimeEnd:datetime "+00:00",
SequenceNumber: "
sequenceNumberStart:string "-" sequenceNumberEnd:string ", Count: "
messageCount:int
| summarize messageCount = sum(messageCount) by cloud_RoleInstance,
bin(timestamp, 5m)
| render areachart kind=stacked

```



Message distribution across instances

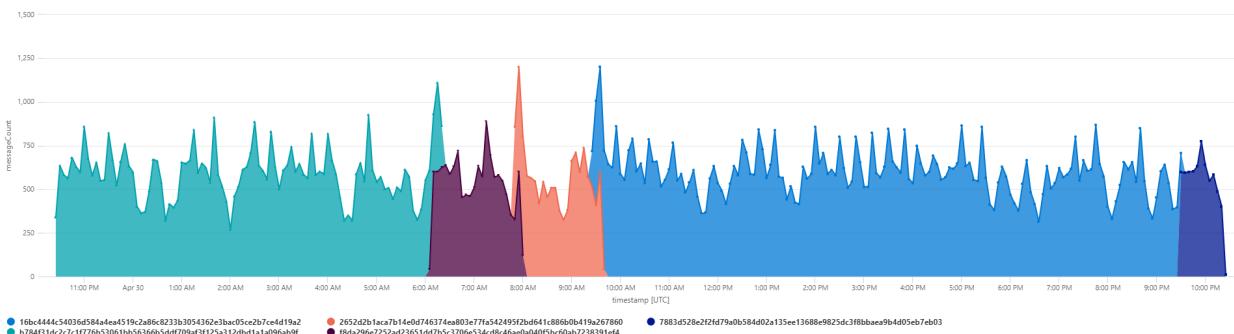
This query shows how to visualize message distribution across instances.

Kusto

```

traces
| where message startswith "Trigger Details: Parti"
| parse message with * "tionId: " partitionId:string ", Offset: "
offsetStart:string "-" offsetEnd:string", EnqueueTimeUtc: "
enqueueTimeStart:datetime "+00:00-" enqueueTimeEnd:datetime "+00:00",
SequenceNumber: "
sequenceNumberStart:string "-" sequenceNumberEnd:string ", Count: "
messageCount:int
| summarize messageCount = sum(messageCount) by cloud_RoleInstance,
bin(timestamp, 5m)
| render areachart kind=stacked

```

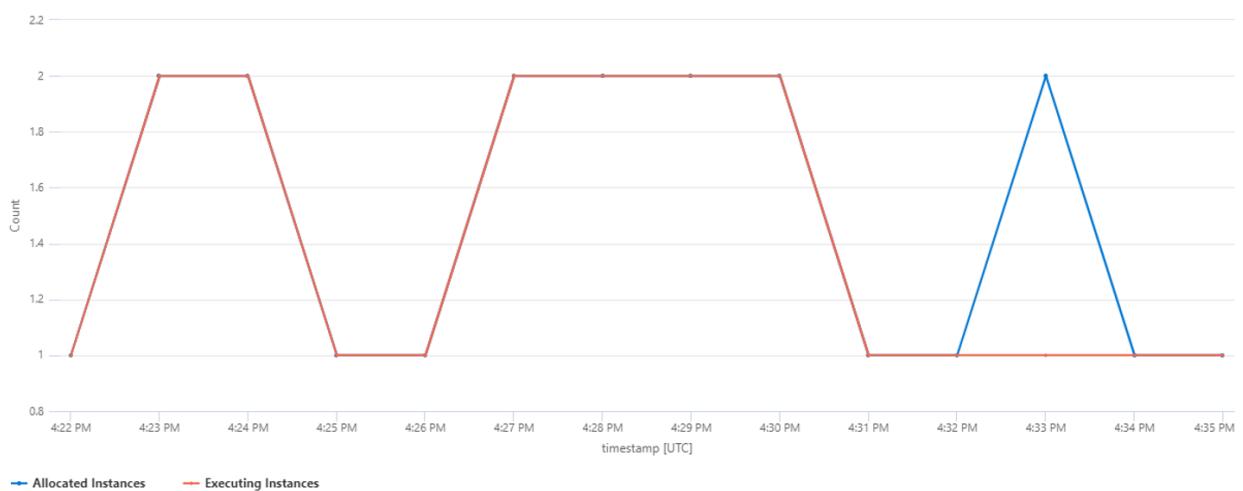


Executing Instances and Allocated Instances

This query shows how to visualize the number of Azure Functions instances that are processing events from Event Hubs, and the total number of instances (processing and waiting for lease). Most of the time they should be the same.

```
Kusto

traces
| where message startswith "Trigger Details: Parti"
| summarize type = "Executing Instances", Count = dcount(cloud_RoleInstance)
by
bin(timestamp, 60s)
| union (
    traces
    | summarize type = "Allocated Instances", Count =
dcount(cloud_RoleInstance) by
bin(timestamp, 60s)
)
| project timestamp, type, Count
| render timechart
```



All Telemetry for a specific Function Execution

The `operation_Id` field can be used across the different tables in Application Insights. For Event Hubs triggered Azure Functions the following query for example will result in the trigger information, telemetry from logs inside the Function code, and dependencies and exceptions:

```
Kusto

union isfuzzy=true requests, exceptions, traces, dependencies
| where * has "<ENTER THE OPERATION_ID OF YOUR FUNCTION EXECUTION HERE>"
| order by timestamp asc
```

timestamp (UTC)	id	name	success	resultCode	duration	performanceBucket	itemType	customDimensions	operation_Name	operation_Id
> 9/9/2021 10:48:30.588 P...	995238e182cbb...	ProcessorFunction	True	0	40.67	<250ms	request	{"FunctionExecutionTimeMs":"40.0062","HostInstanceId":"52c5...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.588 P...	995238e182cbb...	ProcessorFunction	True	0	40.67	<250ms	request	{"FunctionExecutionTimeMs":"40.0062","HostInstanceId":"52c5...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.603 P...							trace	{"prop_{OriginalFormat}":"Executing '{functionName}' (Reason...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.603 P...							trace	{"prop_{OriginalFormat}":"Trigger Details: PartitionId: (PartitionId...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.603 P...							trace	{"prop_{OriginalFormat}":"Executing '{functionName}' (Reason...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.612 PM							trace	{"prop_{OriginalFormat}":"DebatchingFunction: batchSize=1,"...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.612 PM							trace	{"prop_{OriginalFormat}":"DebatchingFunction: batchSize=1,"...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.620 P...							trace	{"prop_{OriginalFormat}":"No sensors matching yellow in this ...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.620 P...							trace	{"prop_{OriginalFormat}":"No sensors matching yellow in this ...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.620 P...							trace	{"prop_{OriginalFormat}":"DebatchingFunction: successfulMes...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.620 P...							trace	{"prop_{OriginalFormat}":"DebatchingFunction: successfulMes...	ProcessorFunction	69429fa63cadbf448
> 9/9/2021, 10:48:30.624 P...							trace	{"prop_{OriginalFormat}":"TransformingFunction: successfulM...	ProcessorFunction	69429fa63cadbf448

End-to-End Latency for an Event

As the `enqueueTimeUtc` property in the trigger detail trace shows the enqueue time of only the first event of each batch that the function processed, a more advanced query can be used to calculate the end-to-end latency of events between two functions with Event Hubs in between. This query will expand the operation links (if any) in the second function's request and map its end time to the same corresponding operation ID of the first function start time.

```
Kusto

let start = view(){
    requests
    | where operation_Name == "FirstFunction"
    | project start_t = timestamp, first_operation_Id = operation_Id
};
let link = view(){
    requests
    | where operation_Name == "SecondFunction"
    | mv-expand ex = parse_json(tostring(customDimensions["_MS.links"]))
    | extend parent = case(isnotempty(ex.operation_Id), ex.operation_Id,
    operation_Id )
    | project first_operation_Id = parent, second_operation_Id = operation_Id
};
let finish = view(){
    traces
    | where customDimensions["EventName"] == "FunctionCompleted" and
    operation_Name
    == "SecondFunction"
    | project end_t = timestamp, second_operation_Id = operation_Id
};
start
| join kind=inner (
link
| join kind=inner finish on second_operation_Id
) on first_operation_Id
| project start_t, end_t, first_operation_Id, second_operation_Id
| summarize avg(datetime_diff('second', end_t, start_t))
```

```

16 | where customDimensions["EventName"] == "FunctionCompleted" and operation_Name == "SecondFunction"
17 | project end_t = timestamp, second_operation_Id = operation_Id
18 };
19 start
20 | join kind=inner (
21 link
22 | join kind=inner finish on second_operation_Id
23 ) on first_operation_Id
24 | project start_t, first_operation_Id, second_operation_Id, end_t
25

```

start_t [Local Time]	first_operation_Id	second_operation_Id	end_t [Local Time]
11/6/2020, 11:04:54.675 AM	7f008be040722b47bee931447c2c60...	967661af22ba54bbe037cdf5cd23aff	11/6/2020, 11:04:54.713 AM
11/6/2020, 11:04:54.675 AM	7f008be040722b47bee931447c2c60...	967661af22ba54bbe037cdf5cd23aff	11/6/2020, 11:04:54.713 AM
11/6/2020, 11:04:54.675 AM	7f008be040722b47bee931447c2c60...	967661af22ba54bbe037cdf5cd23aff	11/6/2020, 11:04:54.713 AM
11/6/2020, 11:04:54.675 AM	7f008be040722b47bee931447c2c60...	967661af22ba54bbe037cdf5cd23aff	11/6/2020, 11:04:54.713 AM
11/6/2020, 11:04:54.675 AM	7f008be040722b47bee931447c2c60...	967661af22ba54bbe037cdf5cd23aff	11/6/2020, 11:04:54.713 AM
11/6/2020, 11:04:55.258 AM	86c7d76f4183e9419293834e2d2898...	8014c4915f25fa4aa0da4662502b1563	11/6/2020, 11:04:55.289 AM
11/6/2020, 11:04:55.258 AM	86c7d76f4183e9419293834e2d2898...	8014c4915f25fa4aa0da4662502b1563	11/6/2020, 11:04:55.289 AM
11/6/2020, 11:04:55.258 AM	86c7d76f4183e9419293834e2d2898...	8014c4915f25fa4aa0da4662502b1563	11/6/2020, 11:04:55.289 AM

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [David Barkol](#) | Principal Solution Specialist GBB

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

To learn more, consider reviewing these related articles:

- [Monitoring serverless event processing](#)
- [Analyze Azure Functions telemetry in Application Insights](#)
- [Configure monitoring for Azure Functions](#)
- [Analyze Azure Functions telemetry in Application Insights](#)
- [Metrics in Azure Monitor - Azure Event Hubs](#)
- [Kusto Query Language](#)

Related resources

- [Monitoring serverless event processing](#) provides guidance on monitoring serverless event-driven architectures.

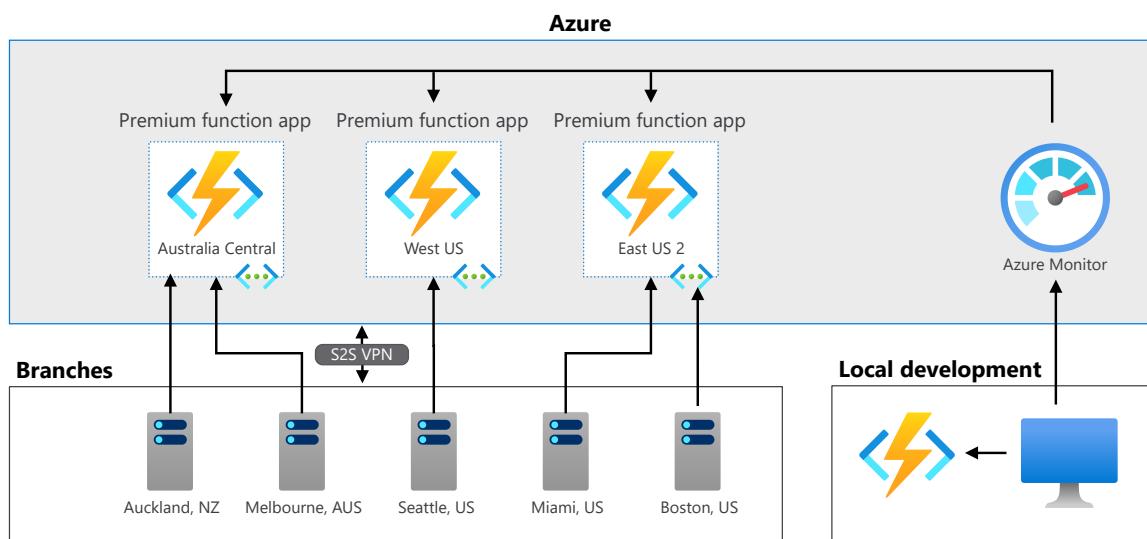
- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the reference architecture work.

Azure Functions in a hybrid environment

Azure Functions Azure Monitor Azure Pipelines Azure Storage Azure Virtual Network

This reference architecture illustrates multiple local branches of an organization that's spread out geographically. Each location is using a Microsoft Azure Function App that's configured with the Premium plan in a nearby cloud region. The developers in this architecture are monitoring all the Azure Function Apps by using Azure Monitor as a single pane of glass.

Architecture



 Microsoft Azure

Download a [Visio file](#) of this architecture.

Components

The architecture consists of the following components:

- **Azure Functions**. Azure Functions is a serverless platform as a service (PaaS) in Azure that runs small, single-task code without requiring new infrastructure to be spun up. The [Azure Functions Premium plan](#) adds the ability to communicate with Azure Functions privately over a virtual network.

- **Azure Virtual Network** [🔗](#). Azure virtual networks are private networks built on the Azure cloud platform so that Azure resources can communicate with each other in a secure fashion. Azure virtual networks [service endpoints](#) ensure that Azure resources can only communicate over the secure virtual network backbone.
- **On-premises network**. In this architecture, the organization has created a secure private network that connects the various branches. This private network is connected to their Azure virtual networks by using a [site-to-site](#) connection.
- **Developer workstations**. In this architecture, individual developers might work on code for Azure Functions entirely on the secure private network or from any remote location. In either scenario, developers have access to Azure Monitor to query or observe metrics and logs for the function apps.

Scenario details

Typical uses for this architecture include:

- Organizations with many physical locations that are connected to a virtual network in Azure to communicate with Azure Functions.
- High-growth workloads that are using Azure Functions locally and maintaining the option to use Azure for any unexpected bursts in work.

Recommendations

The following recommendations apply for most scenarios. Follow these recommendations unless you have a specific requirement that overrides them.

Designing for a serverless architecture

Traditional enterprise applications trend toward a monolithic application architecture in which one code "solution" runs the entire organization's business logic. With Azure Functions, the [best practice](#) is to design for a [serverless architecture](#) in which individual functions perform single tasks. These single tasks are designed to run quickly and integrate into larger workflows.

Serverless architecture on Azure Functions has many benefits, including:

- Applications can [automatically scale](#) by individual business functions instead of scaling the entire solution. This can help keep costs down by scaling only what's needed for each task to serve existing workloads.
- Azure Functions provides [declarative bindings](#) for [many Azure services](#), reducing the amount of code your team needs to write, test, and maintain.

- Individual functions can be reused, reducing the amount of repeated code that's necessary for large enterprise solutions.

Running Azure Functions on-premises

You may choose to have Azure Functions run on-premises rather than in Azure; for example:

- Your team might want to run Azure Functions within an existing on-premises Kubernetes installation.
- In development, your team may find it easier to develop locally using the command line interface instead of the in-portal editor.
- Your functions will run locally with the toolset installed on on-premises VMs.

You can run Azure Functions on-premises in three ways:

- [Azure Functions Core Tools](#). Azure Functions Core Tools is a developer suite that typically [installs from node package manager \(npm\)](#). It allows developers to develop, debug, and test function apps at the command prompt on a local computer.
- [Azure Functions Docker container image](#). You can use this [container image](#) as a base image for containers that run Azure Functions on a Docker host or in Kubernetes.
- [Kubernetes](#). Azure Functions supports [seamless event-driven scale within a Kubernetes cluster](#) using [Kubernetes-based Event Driven Autoscaling \(KEDA\)](#). To review best practices for managing [Azure Kubernetes Service clusters](#) and [Azure Arc-enabled Kubernetes](#) clusters, review the [Run containers in a hybrid environment](#) reference architecture.

Network connectivity

Creating function apps by using the Premium plan opens up the possibility of highly secure [cross-network connectivity](#) between Azure virtual networks, Azure and on-premises networks, and the networks for each on-premises branch.

You should use either a [site-to-site](#) or an [Azure ExpressRoute](#) connection between Azure Virtual Network and on-premises networks. This allows the on-premises branches to communicate with the function apps in Azure by using their [service endpoints](#).

Additionally, each virtual network in Azure should also use [virtual network peering](#) to enable communication between individual function apps across regions.

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Scalability

- Azure Functions code should be designed so that it can scale out endlessly. Consider race conditions, leased files, and other workloads that might cause one function run to block another. Also consider writing all Azure Functions code to be [stateless and defensive](#) in its design.
- For function apps that use [Azure Storage](#) accounts in triggers or bindings, don't reuse the same account that's used to store metadata about the function apps and their runs.

Availability

- Typically, Azure Functions on the consumption plan can scale down to zero instances. When a new event triggers a function app, a new instance must be created with your code running on it. The latency that's associated with this process is referred to as a *cold start*. The Azure Functions Premium plan offers the option to configure [pre-warmed instances](#) that are ready for any new requests. You can configure the number of pre-warmed instances up to the minimum number of instances in your scale-out configuration.
- Consider having multiple Premium plans in multiple regions and using [Azure Traffic Manager](#) to route requests appropriately.

Manageability

- Azure Functions must be in an empty subnet that's a different subnet than your other Azure resources. This might require more planning when designing subnets for your virtual network.
- Consider creating proxies for every on-premises resource that Azure Functions might need to access. This can protect your application integrity against any unanticipated on-premises networking changes.
- Use Azure Monitor to [observe analytics and logs](#) for Azure Functions across your entire solution.

DevOps

- Ideally, deployment operations should come from a single team (*Dev* or *DevOps*), not from each individual branch. Consider using a modern workflow system like Azure Pipelines or GitHub Actions to deploy function apps in a repeatable manner across all Azure regions and potentially on-premises.
- Use your workflow system to automate the redeployment of code to Azure Functions as the code is updated and tagged for release.
- Use [deployment slots](#) to test Azure Functions prior to their final push to production.

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

- Use the [Azure pricing calculator](#) to estimate costs.
- The Azure Functions Premium plan is required for Azure Virtual Network connectivity, private site access, service endpoints, and pre-warmed instances.
- The Azure Functions Premium plan bills on instances instead of consumption. The minimum of a single instance ensures there will be at least some monthly bill even without runs. You can set a maximum instance count to control costs for workloads that may burst in size.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Raunak Jhawar](#) | Senior Cloud Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Azure Functions documentation](#)
- [Azure App Service Hybrid Connections](#)
- [Managing hybrid environments with PowerShell](#)
- [Azure Functions to connect to resources in an Azure virtual network](#)
- [Azure Virtual Network documentation](#)

Related resources

See the following architectural guidance for Azure Functions:

- [Serverless event processing](#)
- [Azure Functions in a hybrid environment](#)
- [Cross-cloud scaling with Azure Functions](#)
- [Code walkthrough: Serverless application with Functions](#)

See the following architectural guidance for Azure Virtual Networks:

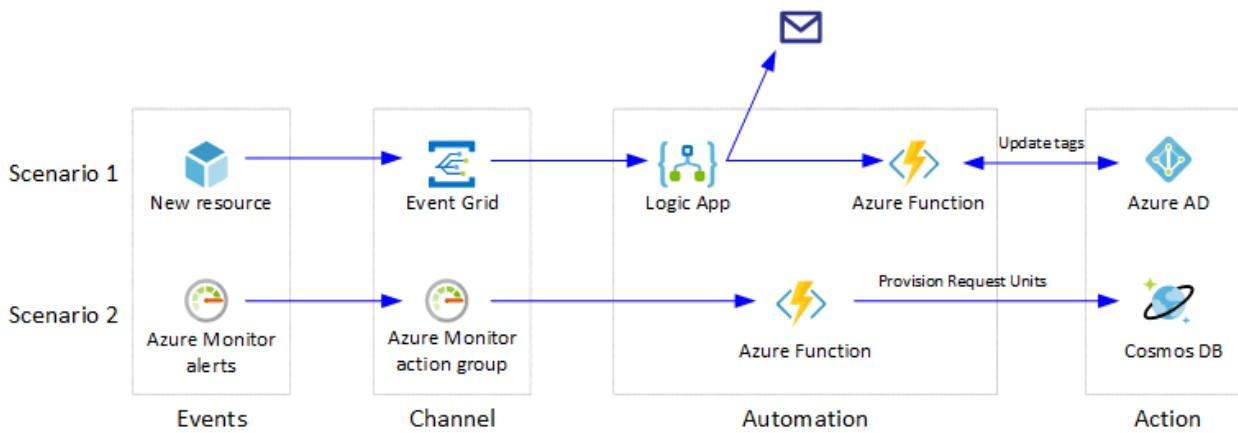
- [Choose between virtual network peering and VPN gateways](#)
- [Virtual network integrated serverless microservices](#)
- [Spoke-to-spoke networking](#)
- [Hub-spoke network topology in Azure](#)
- [Firewall and Application Gateway for virtual networks](#)
- [Deploy AD DS in an Azure virtual network](#)

Event-based cloud automation

Azure Event Grid Azure Functions Azure Logic Apps Azure Monitor Azure Cosmos DB

Automating workflows and repetitive tasks on the cloud, by using [serverless technologies](#), can dramatically improve productivity of an organization's DevOps team. A serverless model is best suited for automation scenarios that fit an [event driven approach](#).

Architecture



Scenarios

This article illustrates two key cloud automation scenarios:

1. **Cost center tagging**: This implementation tracks the cost centers of each Azure resource. The [Azure Policy](#) service [tags all new resources](#) in a group with a default cost center ID. The Azure Event Grid monitors resource creation events, and then calls an [Azure function](#). The function interacts with Microsoft Entra ID, and validates the cost center ID for the new resource. If different, it updates the tag and sends out an email to the resource owner. The REST queries for Microsoft Entra ID are mocked out for simplicity. Microsoft Entra ID can also be integrated using the [Microsoft Graph PowerShell module](#) or the [Microsoft Authentication Library \(MSAL\) for Python](#).
2. **Throttling response**: This example monitors an Azure Cosmos DB database for throttling. [Azure Monitor alerts](#) are triggered when data access requests to Azure Cosmos DB exceed the [capacity in Request Units \(or RUs\)](#). An [Azure Monitor action group](#) is configured to call the automation function in response to these alerts.

The function scales the RUs to a higher value, increasing the capacity and in turn stopping the alerts.

Note

These solutions are not the only away to accomplish these tasks and are shown as illustrative of how serverless technologies can react to environmental signals (events) and influence changes to your environment. Where practical, use platform-native solutions over custom solutions. For example, Azure Cosmos DB natively supports **autoscale throughput** as a native alternative to the Throttling response scenario.



The reference implementation for scenario one is available on [GitHub](#).

The functions in these serverless cloud automation scenarios are often written in PowerShell and Python, two of the most common scripting languages used in system automation. They are deployed using [Azure Functions Core Tools](#) in Azure CLI. Alternatively, you use the [Az.Functions PowerShell cmdlet to deploy and manage Azure Functions](#).

Workflow

Event-based automation scenarios are best implemented using Azure Functions. They follow these common patterns:

- **Respond to events on resources.** These are responses to events such as an Azure resource or resource group getting created, deleted, changed, and so on. This pattern uses [Event Grid](#) to trigger the function for such events. The cost center tagging scenario is an example of this pattern. Other common scenarios include:
 - granting the DevOps teams access to newly created resource groups,
 - sending notification to the DevOps when a resource is deleted, and
 - responding to maintenance events for resources such as Azure Virtual Machines (VMs).
- **Scheduled tasks.** These are typically maintenance tasks executed using [timer-triggered functions](#). Examples of this pattern are:
 - stopping a resource at night, and starting in the morning,
 - reading Blob Storage content at regular intervals, and converting to an Azure Cosmos DB document,
 - periodically scanning for resources no longer in use, and removing them, and
 - automated backups.

- **Process Azure alerts.** This pattern uses the ease of integrating Azure Monitor alerts and action groups with Azure Functions. The function typically takes remedial actions in response to metrics, log analytics, and alerts originating in the applications and the infrastructure. The throttling response scenario is an example of this pattern. Other common scenarios are:
 - truncating the table when SQL Database reaches maximum size,
 - restarting a service in a VM when it is erroneously stopped, and
 - sending notifications if a function is failing.
- **Orchestrate with external systems.** This pattern enables integration with external systems, using [Logic Apps](#) to orchestrate the workflow. [Logic Apps connectors](#) can easily integrate with several third-party services as well as Microsoft services such as Microsoft 365. Azure Functions can be used for the actual automation. The cost center tagging scenario demonstrates this pattern. Other common scenarios include:
 - monitoring IT processes such as change requests or approvals, and
 - sending email notification when automation task is completed.
- **Expose as a *web hook* or API.** Automation tasks using Azure Functions can be integrated into third-party applications or even command-line tools, by exposing the function as a web hook/API using [an HTTP trigger](#). Multiple authentication methods are available in both PowerShell and Python to secure external access to the function. The automation happens in response to the app-specific external events, for example, integration with power apps or GitHub. Common scenarios include:
 - triggering automation for a failing service, and
 - onboarding users to the organization's resources.
- **Create ChatOps interface.** This pattern enables customers to create a chat-based operational interface, and run development and operations functions and commands in-line with human collaboration. This can integrate with the Azure Bot Framework and use Microsoft Teams or Slack commands for deployment, monitoring, common questions, and so on. A ChatOps interface creates a real-time system for managing production incidents, with each step documented automatically on the chat. Read [How ChatOps can help you DevOps better](#) ↗ for more information.
- **Hybrid automation.** This pattern uses the [Azure App Service Hybrid Connections](#) to install a software component on your local machine. This component allows secure access to resources on that machine. The ability to [manage hybrid environments](#) is currently available on Windows-based systems using PowerShell functions. Common scenarios include:

- managing your on-premises machines, and
- managing other systems behind the firewall (for example, an on-premises SQL Server) through a [jump server](#).

Components

The architecture consists of the following components:

- [Azure Functions](#). Azure Functions provide the event-driven, serverless compute capabilities in this architecture. A function performs automation tasks, when triggered by events or alerts. In two scenarios, a function is invoked with an HTTP request. Code complexity should be minimized, by developing the function that is **stateless** and **idempotent**.

Multiple executions of an idempotent function create the same results. To maintain idempotency, the function scaling in the throttling scenario is kept simplistic. In real world automation, make sure to scale up or down appropriately. Read the [Optimize the performance and reliability of Azure Functions](#) for best practices when writing your functions.

- [Azure Logic Apps](#). Logic Apps can be used to perform simpler tasks, easily implemented using [the built-in connectors](#). These tasks can range from email notifications, to integrating with external management applications. To learn how to use Logic Apps with third-party applications, read [basic enterprise integration in Azure](#).

Logic Apps provides a *no code* or *low code* visual designer, and may be used alone in some automation scenarios. Read [this comparison between Azure Functions and Logic Apps](#) to see which service can fit your scenario.

- [Event Grid](#). Event Grid has built-in support for events from other Azure services, as well as custom events (also called *custom topics*). Operational events such as resource creation can be easily propagated to the automation function, using the Event Grid's built-in mechanism.

Event Grid simplifies the event-based automation with a [publish-subscribe model](#), allowing reliable automation for events delivered over HTTPS.

- [Azure Monitor](#). Azure Monitor alerts can monitor for critical conditions, and take corrective action using Azure Monitor action groups. These action groups are easily integrated with Azure Functions. This is useful to watch for and fix any error conditions in your infrastructure, such as database throttling.

- **Automation action.** This broad block represents other services that your function can interact with, to provide the automation functionality. For example, Microsoft Entra ID for tag validation as in the first scenario, or a database to provision as in the second scenario.

Considerations

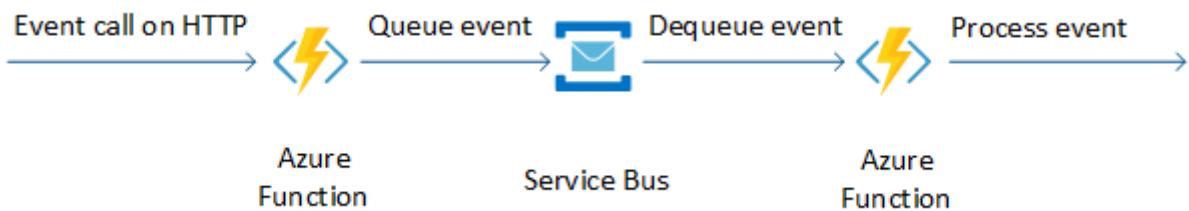
These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Resiliency

Azure Functions

Handle HTTP timeouts

To avoid HTTP timeouts for a longer automation task, queue this event in a [Service Bus](#), and handle the actual automation in another function. The throttling response automation scenario illustrates this pattern, even though the actual Azure Cosmos DB RU provisioning is fast.



[Durable Functions](#), which maintain state between invocations, provide an alternative to the above approach. Durable Functions only support [specific languages](#).

Log failures

As a best practice, the function should log any failures in carrying out automation tasks. This allows for proper troubleshooting of the error conditions. Azure Functions natively supports integration with [Application Insights](#) as the telemetry system.

Concurrency

Verify the concurrency requirement for your automation function. Concurrency is limited by setting the variable `maxConcurrentRequests` in the file [host.json](#). This setting limits the number of concurrent function instances running in your function app. Since every instance consumes CPU and memory, this value needs to be adjusted for CPU-intensive operations. Lower the `maxConcurrentRequests` if your function calls appear to be too slow or aren't able to complete. See the section [Configure host behaviors to better handle concurrency](#) for more details.

Idempotency

Make sure your automation function is idempotent. Both Azure Monitor and Event Grid may emit alerts or events that indicate progression such as your subscribed event is *resolved, fired, in progress, etc.*, your resource is *being provisioned, created successfully, etc.*, or even send false alerts due to a misconfiguration. Make sure your function acts only on the relevant alerts and events, and ignores all others, so that false or misconfigured events do not cause unwanted results. For more information, see [Designing Azure Functions for identical input](#).

Event Grid

If your workflow uses Event Grid, check if your scenario could generate a high volume of events, enough to clog the grid. See [Event Grid message delivery and retry](#) to understand how it handles events when delivery isn't acknowledged, and modify your logic accordingly. The cost center workflow does not implement additional checks for this, since it only watches for resource creation events in a resource group. Monitoring resources created in an entire subscription, can generate larger number of events, requiring a more resilient handling.

Azure Monitor

If a sufficiently large number of alerts are generated, and the automation updates Azure resources, [throttling limits of the Azure Resource Manager](#) might be reached. This can negatively affect the rest of the infrastructure in that subscription. Avoid this situation by limiting the *frequency* of alerts getting generated by the Azure Monitor. You may also limit the alerts getting generated for a particular error. Refer to the [documentation on Azure Monitor alerts](#) for more information.

Security

Security provides assurances against deliberate attacks and the abuse of your valuable data and systems. For more information, see [Overview of the security pillar](#).

Control access to the function

Restrict access to an HTTP-triggered function by setting the [authorization level](#). With *anonymous* authentication, the function is easily accessible with a URL such as

`http://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>`. *Function* level

authentication can obfuscate the http endpoint, by requiring function keys in the URL. This level is set in the file [function.json](#):

```
JSON

{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "Request",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "Response"
    }
  ]
}
```

For production environment, additional strategies might be required to secure the function. In these scenarios, the functions are executed within the Azure platform by other Azure services, and will not be exposed to the Internet. Function authorization is sometimes sufficient for functions accessed as web hooks.

Consider adding security layers on top of function authentication, such as,

- authenticating with client certificates, or
- making sure the caller is part of or has access to the directory that hosts the function, by [enabling App Service Authentication](#).

Function-level authentication is the only option available to Azure Monitor [action groups using Azure Functions](#).

If the function needs to be called from a third-party application or service, it is recommended to provide access to it with an [API Management](#) layer. This layer should enforce authentication. API Management has a [consumption tier](#) integrated with Azure Functions, which allows you to pay only if the API gets executed. For more information, read [Create and expose your functions with OpenAPI](#).

If the calling service supports service endpoints or private link, the following costlier options could be considered:

- Use a dedicated App Service plan, where you can lock down the functions in a virtual network to limit access to it. This is not possible in a consumption-based serverless model.
- Use the [Azure Functions Premium plan](#), which includes a dedicated virtual network to be used by your function apps.

To compare pricing and features between these options, read [Azure Functions scale and hosting](#).

Control what the function can access

[Managed identities for Azure resources](#), a Microsoft Entra feature, simplifies how the function authenticates and accesses other Azure resources and services. The code does not need to manage the authentication credentials, since it is managed by Microsoft Entra ID.

There are two types of managed identities:

- **System-assigned managed identities:** These are created as part of the Azure resource, and cannot be shared among multiple resources. These get deleted when the resource is deleted. Use these for scenarios, which involve single Azure resource or which need independent identities. Both of the scenarios use system-assigned identities since they update only a single resource. Managed identities are only required to update another resource. For example, a function can read the resource tags without a managed identity. See [these instructions](#) to add a system-assigned identity to your function.
- **User-assigned managed identities:** These are created as stand-alone Azure resources. These can be shared across multiple resources, and need to be explicitly deleted when no longer needed. Read [these instructions](#) on how to add user-assigned identity to your function. Use these for scenarios that:
 - Require access to multiple resources that can share a single identity, or
 - Need pre-authorization to secure resources during provisioning, or

- Have resources that are recycled frequently, while permissions need to be consistent.

Once the identity is assigned to the Azure function, assign it a role using [Azure role-based access control \(Azure RBAC\)](#) to access the resources. For example, to update a resource, the *Contributor* role will need to be assigned to the function's identity.

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Use the [Azure pricing calculator](#) to estimate costs. The following are some considerations for lowering cost.

Azure Logic Apps

Logic apps have a pay-as-you-go pricing model. Triggers, actions, and connector executions are metered each time a logic app runs. All successful and unsuccessful actions, including triggers, are considered as executions.

Logic apps have also a fixed pricing model. If you need to run logic apps that communicate with secured resources in an Azure virtual network, you can create them in an [Integration Service Environment \(ISE\)](#).

For details, see [Pricing model for Azure Logic Apps](#).

In this architecture, logic apps are used in the cost center tagging scenario to orchestrate the workflow.

Built-in connectors are used to connect to Azure Functions and send email notification a when an automation task is completed. The functions are exposed as a web hook/API using an HTTP trigger. Logic apps are triggered only when an HTTPS request occurs. This is a cost effective way when compared to a design where functions continuously poll and check for certain criteria. Every polling request is metered as an action.

For more information, see [Logic Apps pricing](#).

Azure Functions

Azure Functions are available with [the following three pricing plans](#).

- **Consumption plan.** This is the most cost-effective, serverless plan available, where you only pay for the time your function runs. Under this plan, functions can run for up to 10 minutes at a time.
- **Premium plan.** Consider using [Azure Functions Premium plan](#) for automation scenarios with additional requirements, such as a dedicated virtual network, a longer execution duration, and so on. These functions can run for up to an hour, and should be chosen for longer automation tasks such as running backups, database indexing, or generating reports.
- **App Service plan.** Hybrid automation scenarios that use the [Azure App Service Hybrid Connections](#), will need to use the App Service plan. The functions created under this plan can run for unlimited duration, similar to a web app.

In these automation scenarios Azure Functions are used for tasks such as updating tags in Microsoft Entra ID, or changing Azure Cosmos DB configuration by scaling up the RUs to a higher value. The **Consumption plan** is the appropriate for this use case because those tasks are interactive and not long-running.

Azure Cosmos DB

Azure Cosmos DB bills for provisioned throughput and consumed storage by hour. Provisioned throughput is expressed in Request Units per second (RU/s), which can be used for typical database operations, such as inserts, reads. Storage is billed for each GB used for your stored data and index. See [Azure Cosmos DB pricing model](#) for more information.

In this architecture, when data access requests to Azure Cosmos DB exceed the capacity in Request Units (or RUs), Azure Monitor triggers alerts. In response to those alerts, an Azure Monitor action group is configured to call the automation function. The function scales the RUs to a higher value. This helps to keep the cost down because you only pay for the resources that your workloads need on a per-hour basis.

To get a quick cost estimate of your workload, use the [Azure Cosmos DB capacity calculator](#).

For more information, see the Cost section in [Microsoft Azure Well-Architected Framework](#).

Deployment considerations

For critical automation workflows that manage behavior of your application, zero downtime deployment must be achieved using an efficient DevOps pipeline. For more

information, read [serverless backend deployment](#).

If the automation covers multiple applications, keep the resources required by the automation in a [separate resource group](#). A single resource group can be shared between automation and application resources, if the automation covers a single application.

If the workflow involves a number of automation functions, group the functions catering to one scenario in a single function app. Read [Manage function app](#) for more information.

As you deploy your application, you will need to monitor it. Consider using [Application Insights](#) to enable the developers to monitor performance and detect issues.

For more information, see the DevOps section in [Microsoft Azure Well-Architected Framework](#).

Imperative actions on Azure resources

In both scenarios above, the end result was a change in Azure resource state via direct Azure Resource Manager interaction. While this was the intended outcome, consider the impact doing so might have on your environment if the modified resources were originally deployed declaratively, such as by Bicep or Terraform templates. Unless those changes are replicated back into your source templates, the next usage of those templates might undo the changes made by this automation. Consider the impact of mixing imperative changes to Azure resources that are routinely deployed via templates.

Deploy this scenario

To deploy the cost center scenario, see the [deployment steps on GitHub](#) ↗.

Next steps

- [Training: Introduction to Azure Functions](#)
- [Documentation: Introduction to Azure Functions](#)
- [What is Azure Event Grid?](#)

Related resources

- [Code walkthrough: Serverless application with Functions](#)
- [Serverless functions architecture design](#)

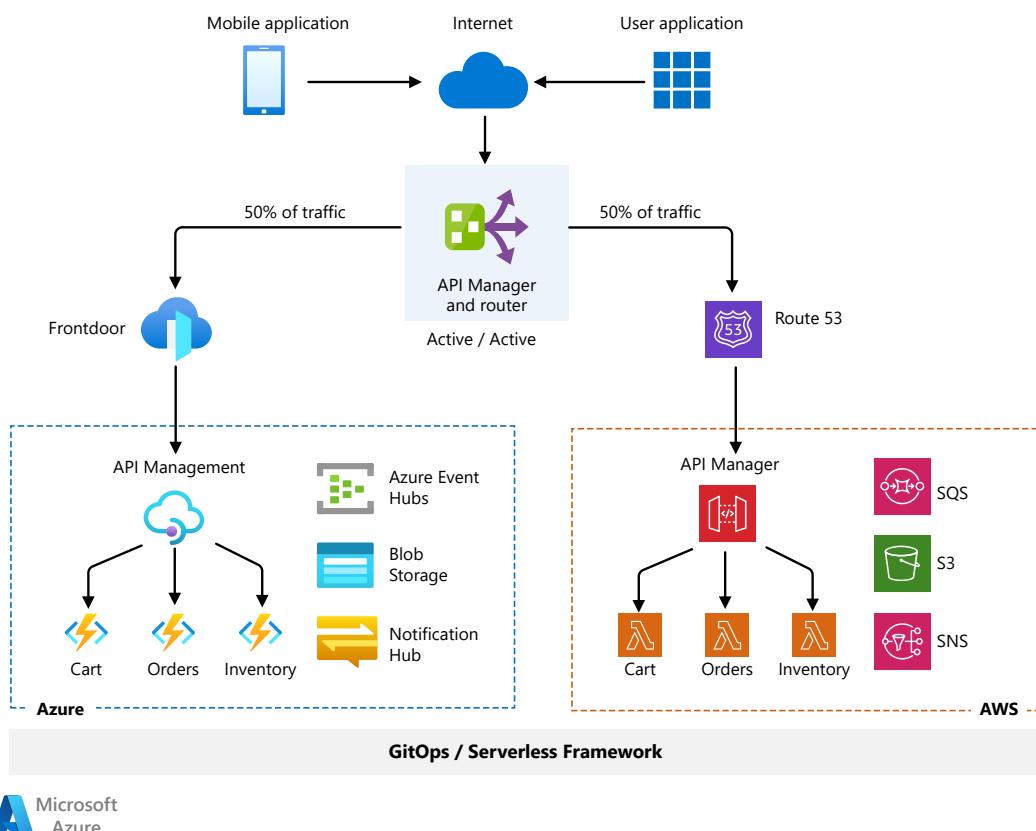
- Serverless functions reference architectures

Multicloud solutions with the Serverless Framework

Azure Functions

This article describes how the Microsoft Commercial Software Engineering (CSE) team partnered with a global retailer to deploy a highly available serverless solution across both Azure and Amazon Web Services (AWS) cloud platforms, using the [Serverless Framework](#).

Architecture



Download a [Visio file](#) of this architecture.

Dataflow

- The user app can come from any source capable of logging into the cloud. In this implementation, the user logs into a gateway app that load balances requests 50-

50 between the Azure and AWS clouds.

- Any response also routes through the API Manager gateway, which then sends it to the requestor app.

Components

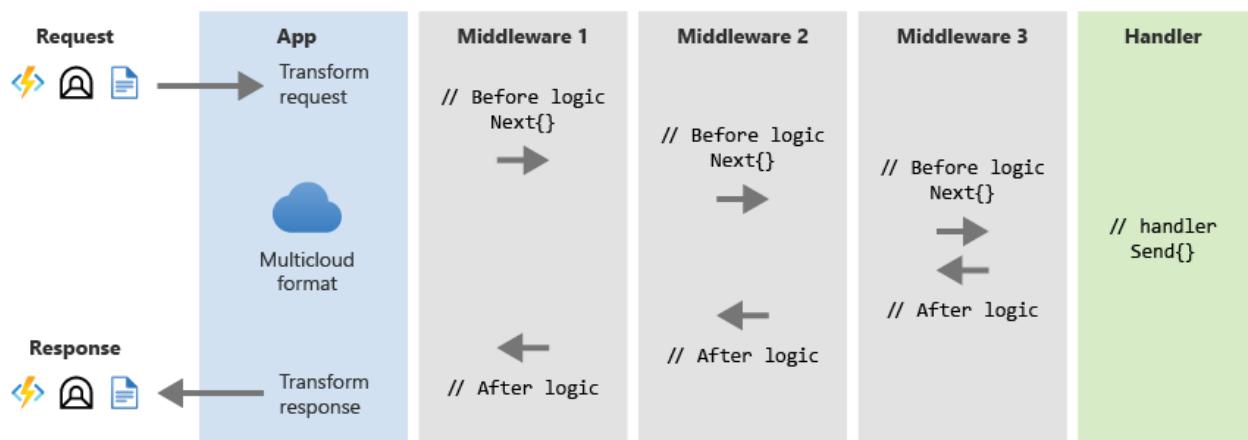
The Serverless Framework

This solution uses the Serverless Framework, available from [Serverless, Inc](#). The free version of the Serverless Framework includes a CLI, more plugins, and limited monitoring services. The Pro edition features operational capabilities across clouds, such as enhanced monitoring and alerts. The framework supports Node.js and Python languages, and both AWS and Azure cloud hosts.

To use Azure with the Serverless Framework, you need:

- Node.js, to package microservices
- Azure Functions, to provide functionality comparable to other cloud platforms
- The Serverless Framework, to support multicloud deployment and monitoring
- The Serverless Multicloud Library, to provide normalized runtime APIs for developers
- The Azure Functions Serverless Plugin, to support multicloud deployment. This plugin wasn't initially up to parity with the comparable AWS Lambda plug-in, and was extended for this project.

The following figure shows the processing pipeline. The middleware layers represent any intermediate functionality needed before reaching the handler.

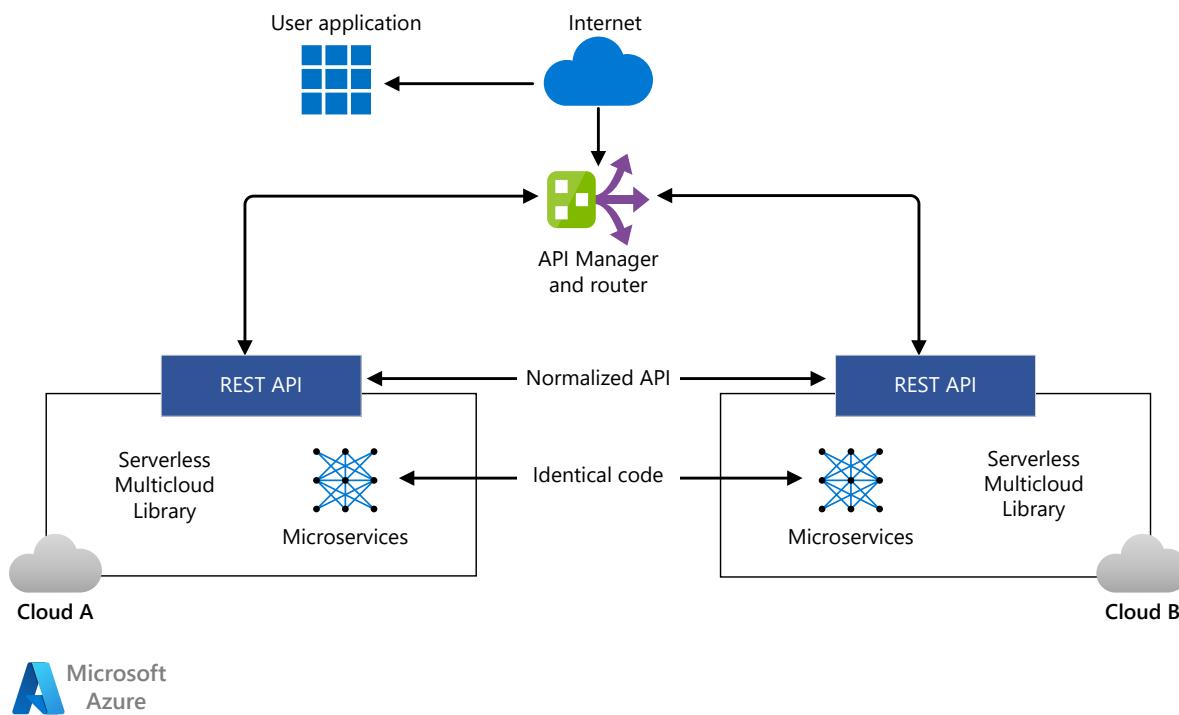


Cloud-agnostic APIs

The serverless implementation on each platform supports individual functions as microservices, one to each functional VM node, and executes processing as needed.

Each AWS Lambda function has a corresponding Azure Functions function. The *Serverless Multicloud Library* builds analogous microservices from either cloud into a cloud-agnostic *normalized REST API* that client apps can use to interface with either platform. Because the abstracted API layer provides code to address the corresponding microservices for each platform, transactions don't need translation. The cloud-agnostic interface lets user apps interact with the cloud without knowing or caring which cloud platform they're accessing.

The following diagram illustrates this concept:



CI/CD with GitOps

A primary job of the Serverless Framework is to abstract away all the infrastructure concerns of deploying an app to the cloud. By using a manifest-based approach, the Serverless Framework can deal with all deployment issues, allowing deployment to be automated as needed to support GitOps.

Although this initial project used manual deployments, it's realistic to implement manifest-driven serverless builds, tests, and deployments within or across clouds. This process can use a GitOps developer workflow: building from Git, using quality gates for test and evaluation, and pushing serverless solutions onto both cloud providers. Performing all deployments using the Serverless Framework from the beginning of the project is the most efficient way to proceed.

API manager

The API Manager can be an existing or custom application. The Apigee™ API Manager in this implementation acted only as a router to provide a 50-50 transaction load balance to the two cloud platforms, and was underutilized for its capabilities.

The API Manager must be able to:

- Be deployed inside or outside a cloud platform as needed
- Route messages to and from both cloud platforms
- Log traffic requests to coordinate asynchronous message traffic
- Relay requests and responses using the common REST API from and to the user application
- Monitor the health of both cloud serverless framework deployments to validate their ability to receive requests
- Perform automated health and availability checks on each cloud platform, to support routing and high availability

Alternatives

- Other languages such as Python could implement the solution, as long as they're supported by the serverless implementations of the cloud platforms, AWS Lambda and Azure Functions in this case. This project used Node.js to package the microservices, because the customer was comfortable with Node.js, and both AWS and Azure platforms support it.
- The solution can use any cloud platform that supports the Serverless Framework, not just Azure and AWS. Currently, the Serverless Framework reports compatibility with eight different cloud providers. The only caveat is to ensure that the elements that support the multicloud architecture or its equivalent are available on the target cloud platforms.
- The API Manager in this initial implementation acted only as a router to provide a 50-50 transaction load balance to the two cloud platforms. The API Manager could incorporate other business logic for specific scenarios.

Scenario details

In *serverless computing*, the cloud provider dynamically allocates microservices resources to run code, and only charges for the resources used. Serverless computing abstracts app code from infrastructure implementation, code deployment, and operational aspects like planning and maintenance.

As with other services, each cloud provider has its own serverless implementation, and it's difficult for customers to use a different provider without considerable operational impact and costs. Potential customers may view this situation as weakening their bargaining position and agility. Vendor lock-in is one of the greatest obstacles to enterprise cloud adoption.

The open-source *Serverless Framework* is a universal cloud interface for developing and deploying serverless computing solutions across cloud providers. Open-sourcing and common APIs for serverless functions help providers, customers, and partners build cross-cloud solutions for best-of-breed services. The Serverless Framework reduces barriers to cloud adoption by addressing the problems of vendor lock-in and cross-cloud provider redundancy. Customers can optimize their solutions based on cost, agility, and other considerations.

CSE and the Azure product team collectively rewrote the *Serverless CLI* to support new Azure Functions features like Premium Functions, API Management, and KeyVault. The Serverless CLI now provides a standard interface for GitOps deployment to both Azure and AWS. The team also developed the *Serverless Multicloud Library*, which provides a *normalized runtime API* to deploy serverless apps to both AWS and Azure.

This design provides high availability with *active-active* failover between multiple cloud platforms, as opposed to *active-passive* failover. If the service of one cloud provider becomes unhealthy or unavailable, this solution can reroute requests to another cloud platform.

This project met the following technical goals:

- Create a cross-industry solution.
- Use the Multicloud Serverless Library to support a cloud-agnostic API that interfaces with microservices wherever they are deployed.
- Support a GitOps CI/CD process workflow for development, testing, and deployment on all supported cloud platforms.
- Use API-based access via an authenticated cloud gateway, and load balance between cloud platforms by using the gateway as a router.

Other potential benefits of using the Serverless Framework include:

- Prevention or reduction of vendor lock-in
- 40-60+% code reduction during development by using the Multicloud Serverless Library
- Development of best-of-breed solutions that combine different cloud providers' services

- Elimination of most platform and infrastructure complexity and maintenance requirements
- Easier data sharing, performance and cost comparisons, and ability to take advantage of special offerings
- Active-active high availability

Potential use cases

- Write client-side applications for multiple platforms by using a cloud-agnostic API from the Serverless Multicloud Library.
- Deploy a collection of functional microservices in a serverless framework to multiple cloud platforms.
- Use a cloud-agnostic app across cloud platforms without knowing or caring which platform is hosting it.

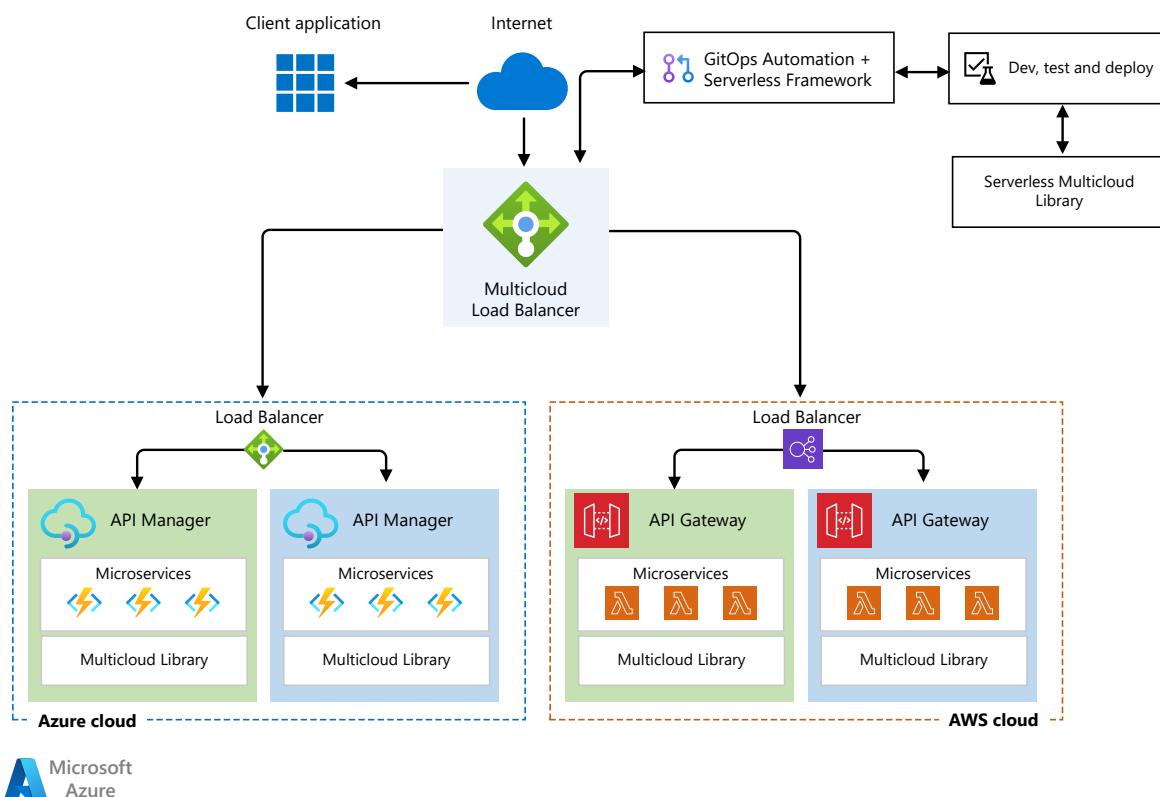
Considerations

- This article doesn't describe security solutions, although the initial deployment included them. There are many possible security solutions, some platform dependent, and this framework should accommodate any reasonable solution. User authentication is the minimum security assumed.
- Because it's difficult to articulate the differences between AWS and Azure serverless functional offerings, early effort should focus on mapping the functions available on each cloud platform and identifying necessary transformation requirements. You can develop a platform-agnostic API from this information.
- Using an open-source solution may introduce risks, due to long-term maintenance and support challenges with any open-source software.
- In the free Serverless Framework, monitoring is limited primarily to the administrative dashboard. Monitoring is available in the paid enterprise offering. Currently, the Azure Functions Serverless Plugin doesn't include provisions for observability or monitoring, and would need modification to implement these capabilities.
- This solution could use metrics to compare performance and costs between cloud platforms, enabling customers to seamlessly optimize usage across cloud platforms.

Deploy this scenario

A traditional *Blue-Green Deployment* develops and deploys an app to two separate but identical environments, blue and green, increasing availability and reducing risk. The blue environment is usually the production environment that normally handles live traffic, and the green environment is a failover deployment as needed. Typically, the CI/CD pipeline automatically deploys both blue and green environments within the same cloud platform. This configuration is considered an *active-passive* configuration.

In the multicloud solution, blue-green deployment is implemented in both cloud platforms. In the serverless case, two duplicate sets of microservices are deployed for each cloud platform, one as the production environment and the other as the failover environment. This active-passive setup within each cloud platform reduces the risk that this platform will be down, increasing its availability, and enabling multicloud *active-active* high availability.



A secondary benefit of blue-green deployment is the ability to use the failover deployment on each cloud platform as a test environment for microservices updates, before releasing them to the production deployment.

Next steps

- Sample code [↗](#) and [README ↗](#) for this implementation on GitHub
- [Serverless Framework ↗](#)
- [Training: Introduction to Azure Functions](#)

- Overview of Azure Functions

Related resources

- [Serverless functions architecture design](#)
- [Serverless functions reference architectures](#)
- [Code walkthrough: Serverless application with Functions](#)

Share a location in real time by using low-cost serverless Azure services

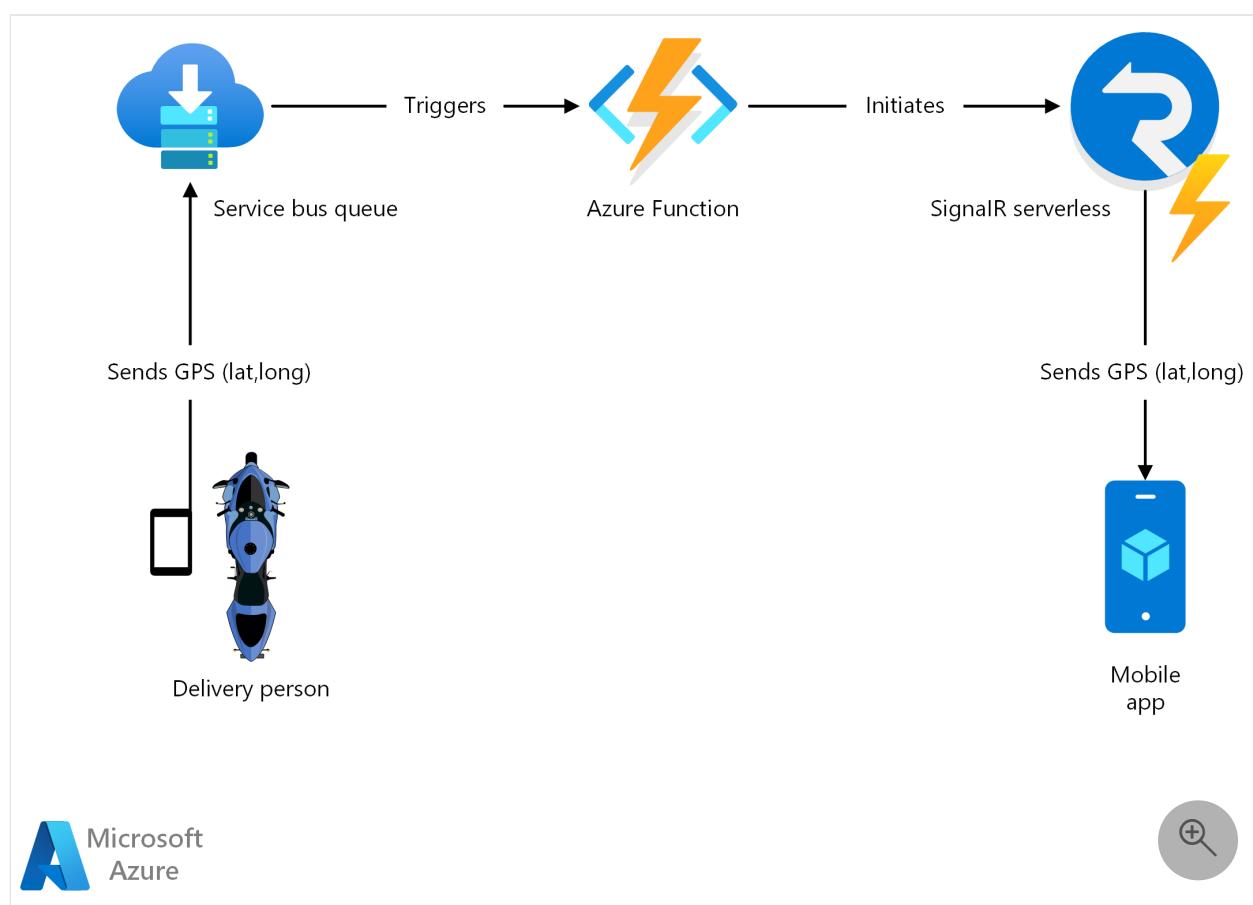
Azure Front Door

Azure Functions

Azure Service Bus

This scenario describes how to architect a solution that processes changes to underlying data within a web view, without the need for a page refresh, by using real-time services. Examples that use this scenario include real-time tracking of products and goods, and social media solutions.

Architecture



Download a [Visio file](#) of this architecture.

Components

- [Azure Service Bus](#) is a highly reliable cloud messaging service between applications and services, even when one or more are offline.

- [Azure SignalR Service](#) makes it easy to add real-time communications to your web application.
- [Azure Functions](#) is an event-driven, serverless compute platform that can also solve complex orchestration problems.

Alternatives

Alternatives exist to address this scenario, including [Pusher](#). It's the category leader in robust APIs for app developers who build scalable real-time communication features.

There's also [PubNub](#). PubNub makes it easy for you to add real-time capabilities to your apps, without worrying about the infrastructure. Build apps that allow your users to engage in real time across mobile, browser, desktop, and server.

[Ably](#) is another alternative. It provides serverless publish/subscribe (pub/sub) messaging, which scales reliably with your needs. The messaging is delivered at the edge using WebSockets. The Ably platform provides a highly available, elastically scalable, and globally distributed real-time infrastructure, at the call of an API.

Although Pusher, PubNub, and Ably are the most widely adopted platforms for real-time messaging, for this scenario, you'll do everything in Azure. We recommend SignalR as the go-to platform, because it allows bi-directional communication between server and client. It's also an open-source tool, with 7.9 thousand GitHub stars and 2.2 thousand GitHub forks.

For more information, see the [SignalR open-source repository](#) on GitHub.

Scenario details

In this scenario, you'll look at how to set up a real-time messaging service to share the live location of a food delivery service transaction. This example can also be useful for those trying to build a real-time location-sharing platform for their web or mobile applications.

You'll use a SignalR service configured in serverless mode to integrate with an Azure Functions app that's triggered by a service bus, all of it by using .NET Core.

Potential use cases

These other use cases have similar design patterns:

- Sharing real-time location with client devices

- Pushing notifications to users
- Updating timelines
- Creating chat rooms

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that you can use to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Here are a couple of things to keep in mind as you develop this scenario, including how to configure parameters in the Azure Service Bus connection string in ServiceBusTrigger:

- **Hubs:** Hubs can be compared to a video streaming service. You can subscribe to a hub to send and receive messages from and to it.
- **Targets:** Targets are like radio channels. They include everyone who's listening to the target channel, and they're notified when there's a new message on it.

If you can remember the preceding two features of the SignalR platform, it will be easy to get up and running quickly.

Availability, scalability, and security

You can achieve high availability with this solution by following what's described in the next two sections.

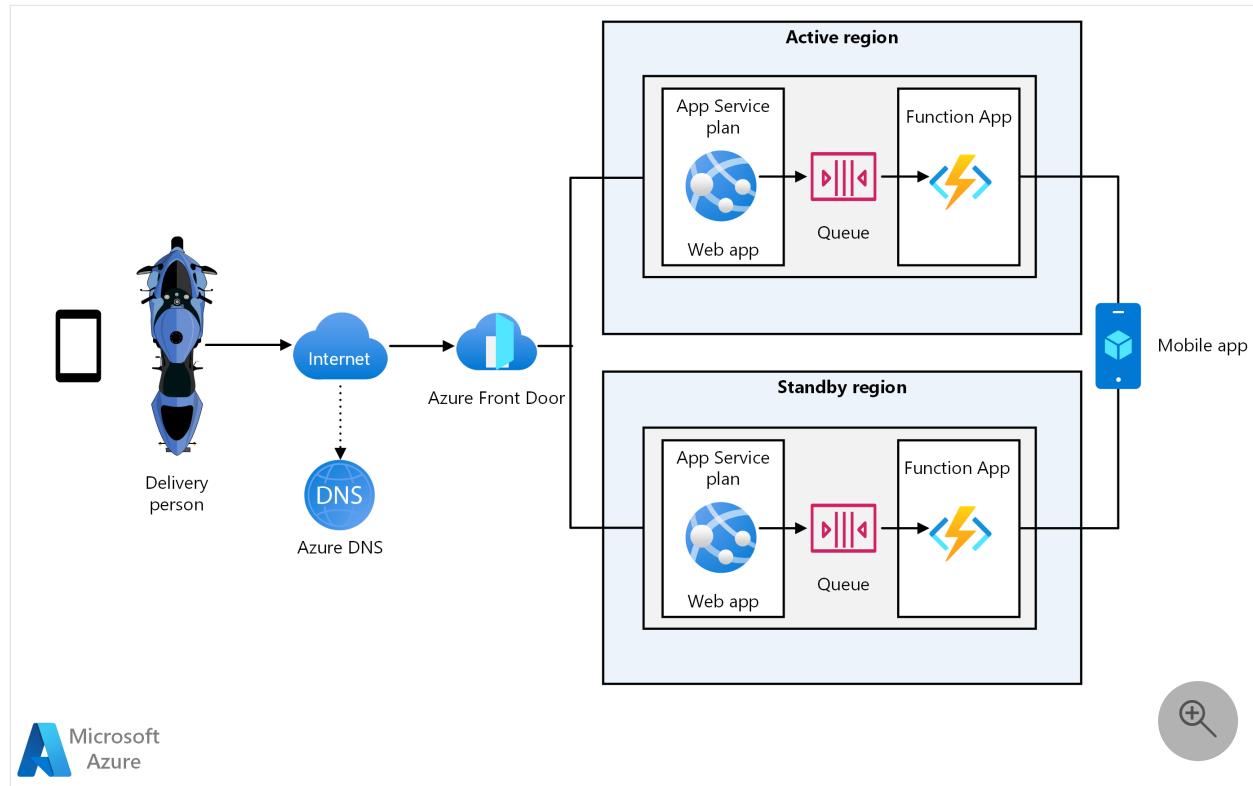
Regional pairing

Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair (for example, East US 2 and Central US). Benefits of doing so include:

- If there's a broad outage, recovery of at least one region out of every pair is prioritized.
- Planned Azure system updates are rolled out to paired regions sequentially to minimize possible downtime.
- In most cases, regional pairs reside within the same geography to meet data residency requirements.
- However, make sure that both regions support all the Azure services that are needed for your application. See [Services by region](#). For more information about

regional pairs, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).

Azure Front Door



[Download a Visio file](#) of this architecture.

Azure Front Door is a scalable and secure entry point for fast delivery of your global applications. When you use *priority routing*, it automatically fails over if the primary region becomes unavailable. A multi-region architecture can provide higher availability than deploying to a single region. If a regional outage affects the primary region, you can use Front Door to fail over to the secondary region.

This architecture can also help if an individual subsystem of the solution fails. Stop network and application layer attacks at the edge with Web Application Firewall and DDoS Protection. Harden your service by using Microsoft-managed rule sets, and author your own rules for custom protection of your app.

Front Door is a possible failure point in the system. If the service fails, clients can't access your application during the downtime. Review the [Front Door service-level agreement \(SLA\)](#) and determine whether using Front Door alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a fallback. If the Front Door service fails, change your canonical name (CNAME) records in DNS to point to the other traffic management service. You must

perform this step manually, and your application will be unavailable until the DNS changes are propagated.

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Let's assume that your business has 1,000 orders a day and needs to share location data with all of them concurrently. Your estimated Azure usage for deploying this scenario will be close to USD192 per month, based on pricing at the date of publication.

Expand table

Service type	Estimated monthly cost
Azure Functions	USD119.40
Azure SignalR Service	USD48.97
Azure Service Bus	USD23.71
Total	USD192.08

Deploy this scenario

Azure Functions development

A serverless real-time application that's built with Azure Functions and Azure SignalR Service ordinarily requires two Azure Functions:

- A `negotiate` function that the client calls to obtain a valid SignalR Service access token and service endpoint URL.
- One or more functions that send messages or manage group membership.

SignalRFunctionApp

SignalRFunctionApp is a function app that creates an Azure Functions instance, with a service bus trigger with SignalR.

Negotiate.cs

This function is triggered by an HTTP request. It's used by client applications to get a token from the SignalR service, which clients can use to subscribe to a hub. This function should be named `negotiate`. For more information, see [Azure Functions development and configuration with Azure SignalR Service](#),

Message.cs

This function is triggered by a service bus trigger. It has a binding with SignalR service. It pulls the message from the queue and passes it on to a SignalR hub.

Instructions

Before you begin:

- Make sure that you have a service bus queue provisioned on Azure.
 - Make sure that you have a SignalR service provisioned in serverless mode on Azure.
1. Enter your connection strings (Service Bus and SignalR) in the `local.settings.json` file.
 2. Enter the URL of the client application (SignalR client) in CORS (Cross-Origin Resource Sharing). For the most recent syntax, see [Azure Functions development and configuration with Azure SignalR Service](#).
 3. Enter your service bus queue name in the service bus trigger in the `Message.cs` file.

Now, let's configure the client application to test it. First, grab the example sources from the [solution-architectures](#) GitHub page.

SignalR client

This is a simple .NET Core web application to subscribe to the hub that's created by SignalRFunctionApp. It displays messages that are received on the service bus queue in real time. Although you can use SignalRFunctionApp to work with a mobile client, let's stick to the web client for this scenario in this repository.

Instructions

1. Make sure that SignalRFunctionApp is running.
2. Copy the URL that's generated by the negotiate function. It looks something like this: `http://localhost:7071/api/`.
3. Paste the URL in the `chat.js` file, inside
`signalR.HubConnectionBuilder().withUrl("YOUR_URL_HERE").build();`.
4. Run the application.

The status is connected when the web client successfully subscribes to the SignalR hub.

SendToQueue.js

This node.js script pushes a message to the Service Bus, so that you can test the deployment you've just completed.

Instructions

1. Install the node Azure Service Bus module (@azure/service-bus).
2. Enter your connection strings and queue name in the script.
3. Run the script.

Next steps

You can take this scenario into your production environment. However, make sure that your Azure services are set to scale. For instance, your Azure Service Bus should be set to a standard or premium plan.

You can deploy the code to Azure Functions right from Visual Studio. To learn how to publish your code to Azure Functions from Visual Studio, follow the [It's how you make software](#) guide.

See [how to work with Azure Service Bus bindings in Azure Functions](#). Azure Functions supports trigger and output bindings for service bus queues and topics.

See [how to authenticate and send real-time messages](#) to clients that are connected to Azure SignalR Service, by using SignalR Service bindings in Azure Functions. Azure Functions supports input and output bindings for SignalR Service.

Related resources

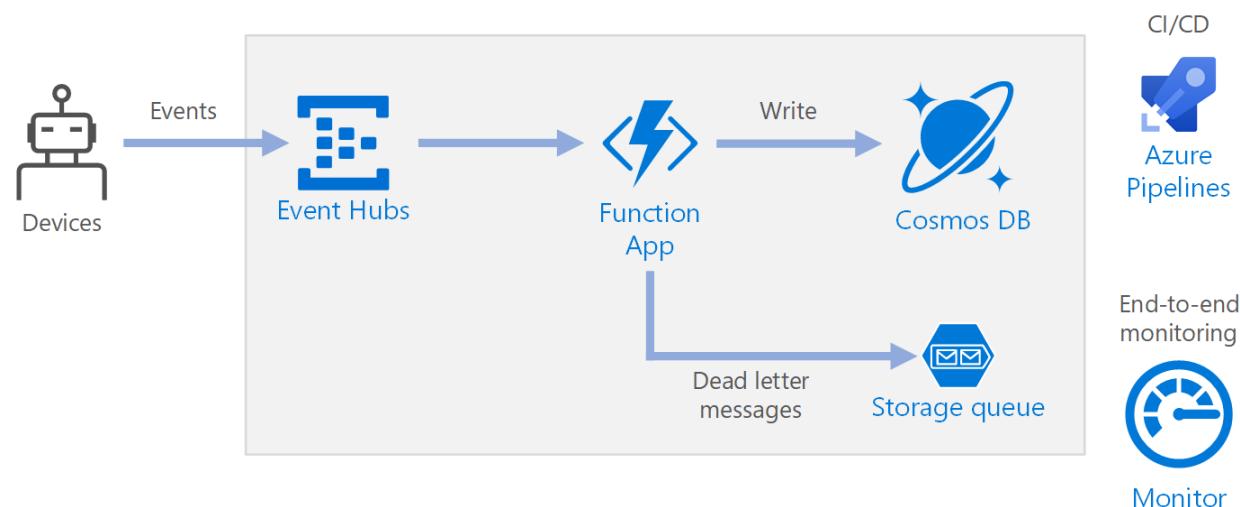
- [High-volume batch transaction processing](#)
- [Transit hub dynamic pub-sub messaging system](#)
- [Serverless event processing](#)
- [Blockchain workflow application](#)
- [Event-based cloud automation](#)

Serverless event processing

Azure Cosmos DB Azure Functions Azure Monitor Azure Pipelines Azure Storage

This reference architecture shows a [serverless](#), event-driven architecture that ingests a stream of data, processes the data, and writes the results to a back-end database.

Architecture



Workflow

- Events arrive at Azure Event Hubs.
- A Function App is triggered to handle the event.
- The event is stored in an Azure Cosmos DB database.
- If the Function App fails to store the event successfully, the event is saved to a Storage queue to be processed later.

Components

- **Event Hubs** ingests the data stream. [Event Hubs](#) is designed for high-throughput data streaming scenarios.

! Note

For Internet of Things (IoT) scenarios, we recommend [Azure IoT Hub](#). IoT Hub has a built-in endpoint that's compatible with the Azure Event Hubs API, so you can use either service in this architecture with no major changes in the

back-end processing. For more information, see [Connecting IoT Devices to Azure: IoT Hub and Event Hubs](#).

- **Function App.** [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (*a function*) is invoked by a trigger. In this architecture, when events arrive at Event Hubs, they trigger a function that processes the events and writes the results to storage.

Function Apps are suitable for processing individual records from Event Hubs. For more complex stream processing scenarios, consider [Apache Spark using Azure Databricks](#), or [Azure Stream Analytics](#).

- **Azure Cosmos DB.** [Azure Cosmos DB](#) is a multi-model database service that is available in a serverless, consumption-based mode. For this scenario, the event-processing function stores JSON records, using [Azure Cosmos DB for NoSQL](#).
- **Queue storage.** [Queue storage](#) is used for dead-letter messages. If an error occurs while processing an event, the function stores the event data in a dead-letter queue for later processing. For more information, see the [Resiliency section](#) later in this article.
- **Azure Monitor.** [Monitor](#) collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get visibility into the health of the solution.
- **Azure Pipelines.** [Pipelines](#) is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that can be used to improve the quality of a workload. For more information, see [Microsoft Azure Well-Architected Framework](#).

Availability

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of geo-distribution features in the various services:

- **Event Hubs.** Create two Event Hubs namespaces, a primary (active) namespace and a secondary (passive) namespace. Messages are automatically routed to the active

namespace unless you fail over to the secondary namespace. For more information, see [Azure Event Hubs Geo-disaster recovery](#).

- **Function App.** Deploy a second function app that is waiting to read from the secondary Event Hubs namespace. This function writes to a secondary storage account for a dead-letter queue.
- **Azure Cosmos DB.** Azure Cosmos DB supports [multiple write regions](#), which enables writes to any region that you add to your Azure Cosmos DB account. If you don't enable multi-write, you can still fail over the primary write region. The Azure Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.
- **Azure Storage.** Use [RA-GRS storage](#) for the dead-letter queue. This creates a read-only replica in another region. If the primary region becomes unavailable, you can read the items currently in the queue. In addition, provision another storage account in the secondary region that the function can write to after a fail-over.

Scalability

Event Hubs

The throughput capacity of Event Hubs is measured in [throughput units](#). You can autoscale an event hub by enabling [auto-inflate](#), which automatically scales the throughput units based on traffic, up to a configured maximum.

The [Event Hubs trigger](#) in the function app scales according to the number of partitions in the event hub. Each partition is assigned one function instance at a time. To maximize throughput, receive the events in a batch, instead of one at a time.

Azure Cosmos DB

Azure Cosmos DB is available in two different capacity modes:

- [Serverless](#), for workloads with intermittent or unpredictable traffic and low average-to-peak traffic ratio.
- [Provisioned throughput](#), for workloads with sustained traffic requiring predictable performance.

To make sure your workload is scalable, it is important to choose an appropriate [partition key](#) when you create your Azure Cosmos DB containers. Here are some characteristics of a good partition key:

- The key value space is large.
- There will be an even distribution of reads/writes per key value, avoiding hot keys.
- The maximum data stored for any single key value won't exceed the maximum physical partition size (20 GB).
- The partition key for a document won't change. You can't update the partition key on an existing document.

In the scenario for this reference architecture, the function stores exactly one document per device that is sending data. The function continually updates the documents with the latest device status using an [upsert operation](#). Device ID is a good partition key for this scenario because writes will be evenly distributed across the keys, and the size of each partition will be strictly bounded because there is a single document for each key value. For more information about partition keys, see [Partition and scale in Azure Cosmos DB](#).

Resiliency

When using the Event Hubs trigger with Functions, catch exceptions within your processing loop. If an unhandled exception occurs, the Functions runtime doesn't retry the messages. If a message can't be processed, put the message into a dead-letter queue. Use an out-of-band process to examine the messages and determine corrective action.

The following code shows how the ingestion function catches exceptions and puts unprocessed messages onto a dead-letter queue.

C#

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public static async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection",
    ConsumerGroup = "%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage>
    deadLetterMessages,
    ILogger logger)
{
    foreach (var message in messages)
    {
        DeviceState deviceState = null;

        try
        {
            deviceState = telemetryProcessor.Deserialize(message.Body.Array,
logger);
        }
        catch (Exception ex)
```

```

    {
        logger.LogError(ex, "Error deserializing message",
message.SystemProperties.PartitionKey,
message.SystemProperties.SequenceNumber);
        await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue
= ex.Message, EventData = message });
    }

    try
    {
        await stateChangeProcessor.UpdateState(deviceState, logger);
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Error updating status document",
deviceState);
        await deadLetterMessages.AddAsync(new DeadLetterMessage { Issue
= ex.Message, EventData = message, DeviceState = deviceState });
    }
}
}

```

Notice that the function uses the [Queue storage output binding](#) to put items in the queue.

The code shown also logs exceptions to Application Insights. You can use the partition key and sequence number to correlate dead-letter messages with the exceptions in the logs.

Messages in the dead-letter queue should have enough information so that you can understand the context of the error. In this example, the `DeadLetterMessage` class contains the exception message, the original event data, and the deserialized event message (if available).

C#

```

public class DeadLetterMessage
{
    public string Issue { get; set; }
    public EventData EventData { get; set; }
    public DeviceState DeviceState { get; set; }
}

```

Use [Azure Monitor](#) to monitor the event hub. If you see there is input but no output, it means that messages aren't being processed. In that case, go into [Log Analytics](#) and look for exceptions or other errors.

Use infrastructure as code (IaC) when possible. IaC manages the infrastructure, application, and storage resources with a declarative approach like [Azure Resource Manager](#). That will help in automating deployment using DevOps as a continuous integration and continuous delivery (CI/CD) solution. Templates should be versioned and included as part of the release pipeline.

When creating templates, group resources as a way to organize and isolate them per workload. A common way to think about workload is a single serverless application or a virtual network. The goal of workload isolation is to associate the resources to a team, so that the DevOps team can independently manage all aspects of those resources and perform CI/CD.

This architecture includes steps to configure the Drone Status Function App using Azure Pipelines with YAML and Azure Functions Slots.

As you deploy your services you will need to monitor them. Consider using [Application Insights](#) to enable the developers to monitor performance and detect issues.

For more information, see the [DevOps checklist](#).

Disaster recovery

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of geo-distribution features in the various services:

- **Event Hubs.** Create two Event Hubs namespaces, a primary (active) namespace and a secondary (passive) namespace. Messages are automatically routed to the active namespace unless you fail over to the secondary namespace. For more information, see [Azure Event Hubs Geo-disaster recovery](#).
- **Function App.** Deploy a second function app that is waiting to read from the secondary Event Hubs namespace. This function writes to a secondary storage account for dead-letter queue.
- **Azure Cosmos DB.** Azure Cosmos DB supports [multiple write regions](#), which enables writes to any region that you add to your Azure Cosmos DB account. If you don't enable multi-write, you can still fail over the primary write region. The Azure Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.

- **Azure Storage.** Use [RA-GRS](#) storage for the dead-letter queue. This creates a read-only replica in another region. If the primary region becomes unavailable, you can read the items currently in the queue. In addition, provision another storage account in the secondary region that the function can write to after a fail-over.

Cost optimization

Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see [Overview of the cost optimization pillar](#).

Use the [Azure Pricing calculator](#) to estimate costs. Here are some other considerations for Azure Functions and Azure Cosmos DB.

Azure Functions

Azure Functions supports two hosting models:

- **Consumption plan.** Compute power is automatically allocated when your code is running.
- **App Service plan.** A set of virtual machines (VMs) are allocated for your code. The App Service plan defines the number of VMs and the VM size.

In this architecture, each event that arrives on Event Hubs triggers a function that processes that event. From a cost perspective, the recommendation is to use the **consumption plan** because you pay only for the compute resources you use.

Azure Cosmos DB

With Azure Cosmos DB, you pay for the operations you perform against the database and for the storage consumed by your data.

- **Database operations.** The way you get charged for your database operations depends on the type of Azure Cosmos DB account you're using.
 - In [serverless mode](#), you don't have to provision any throughput when creating resources in your Azure Cosmos DB account. At the end of your billing period, you get billed for the amount of [Request Units](#) consumed by your database operations.
 - In [provisioned throughput](#) mode, you specify the throughput that you need in [Request Units](#) per second (RU/s), and get billed hourly for the maximum provisioned throughput for a given hour. **Note:** Because the provisioned throughput model dedicates resources to your container or database, you'll be

charged for the throughput you've provisioned even if you don't run any workloads.

- **Storage.** You're billed a flat rate for the total amount of storage (in GBs) consumed by your data and indexes for a given hour.

In this reference architecture, the function stores exactly one document per device that is sending data. The function continually updates the documents with latest device status, using an upsert operation, which is cost effective in terms of consumed storage. For more information, see [Azure Cosmos DB pricing model](#).

Use the [Azure Cosmos DB capacity calculator](#) to get a quick estimate of the workload cost.

Deploy this scenario



A reference implementation for this architecture is [available on GitHub](#).

Next steps

- [Introduction to Azure Functions](#)
- [Welcome to Azure Cosmos DB](#)
- [What is Azure Queue Storage?](#)
- [Azure Monitor overview](#)
- [Azure Pipelines documentation](#)

Related resources

- [Code walkthrough: Serverless application with Azure Functions](#)
- [Monitoring serverless event processing](#)
- [De-batching and filtering in serverless event processing with Event Hubs](#)
- [Private link scenario in event stream processing](#)
- [Azure Kubernetes in event stream processing](#)

Azure Kubernetes in event stream processing

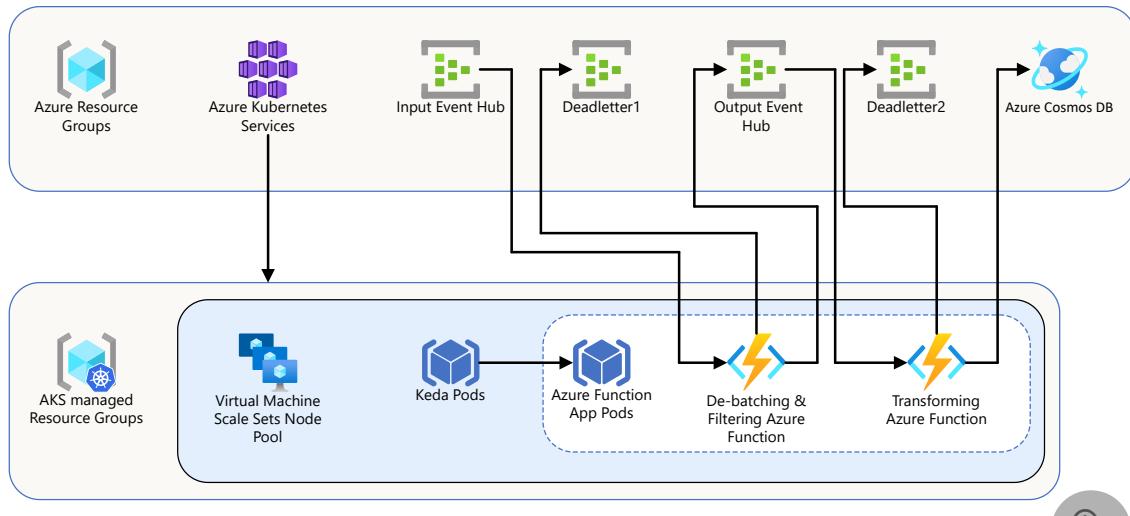
Azure Kubernetes Service (AKS) Azure IoT Hub Azure Event Hubs Azure Functions Azure Cosmos DB

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes a variation of a [serverless](#) event-driven architecture that runs on Azure Kubernetes Service (AKS) with KEDA scaler. The solution ingests a stream of data, processes the data, and then writes the results to a back-end database.

Architecture



 Microsoft Azure



[Download a Visio file](#) of this architecture.

Dataflow

1. AKS with the KEDA scaler is used to autoscale Azure Functions containers based on the number of events needing to be processed.
2. Events arrive at the Input Event Hub.

3. The De-batching and Filtering Azure Function is triggered to handle the event. This step filters out unwanted events and de-batches the received events before submitting to the Output Event Hub.
4. If the De-batching and Filtering Azure Function fails to store the event successfully, the event is submitted to the Deadletter Event Hub 1.
5. Events arriving at the Output Event Hub trigger the Transforming Azure Function. This Azure Function transforms the event into a message for the Azure Cosmos DB instance.
6. The event is stored in an Azure Cosmos DB database.

Components

- [Azure Kubernetes Service](#) (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.
- [KEDA](#) is an event-driven autoscaler used to scale containers in the Kubernetes cluster based on the number of events needing to be processed.
- [Event Hubs](#) ingests the data stream. Event Hubs is designed for high-throughput data streaming scenarios.
- [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (a *function*) is invoked by a trigger.
- [Azure Cosmos DB](#) is a multi-model database service that is available in a serverless, consumption-based mode. For this scenario, the event-processing function stores JSON records, using the [Azure Cosmos DB for NoSQL](#).

ⓘ Note

For Internet of Thing (IoT) scenarios, we recommend [Azure IoT Hub](#). IoT Hub has a built-in endpoint that's compatible with the Azure Event Hubs API, so you can use either service in this architecture with no major changes in the back-end processing. For more information, see [Connecting IoT Devices to Azure: IoT Hub and Event Hubs](#).

Scenario details

This article describes a [serverless](#) event-driven architecture that runs on AKS with KEDA scaler. The solution ingests a stream of data, processes the data, and then writes the results to a back-end database.

To learn more about the basic concepts, considerations, and approaches for serverless event processing, see the [Serverless event processing](#) reference architecture.

Potential use case

A popular use case for implementing an end-to-end event stream processing pattern includes the Event Hubs streaming ingestion service to receive and process events per second using a de-batching and transformation logic implemented with highly scalable, event hub-triggered functions.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rajasa Savant](#) | Senior Software Development Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Introduction to Azure Kubernetes Service](#)
- [Azure Event Hubs documentation](#)
- [Introduction to Azure Functions](#)
- [Azure Functions documentation](#)
- [Overview of Azure Cosmos DB](#)
- [Choose an API in Azure Cosmos DB](#)

Related resources

- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [Monitoring serverless event processing](#) provides an overview and guidance on monitoring serverless event-driven architectures like this one.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the architecture work.
- [Private link scenario in event stream processing](#) is a solution idea for implementing a similar architecture in a virtual network with private endpoints, in order to

enhance security.

Big data analytics with Azure Data Explorer

Azure Data Explorer Azure Event Hubs Azure IoT Hub Azure Storage Azure Synapse Analytics

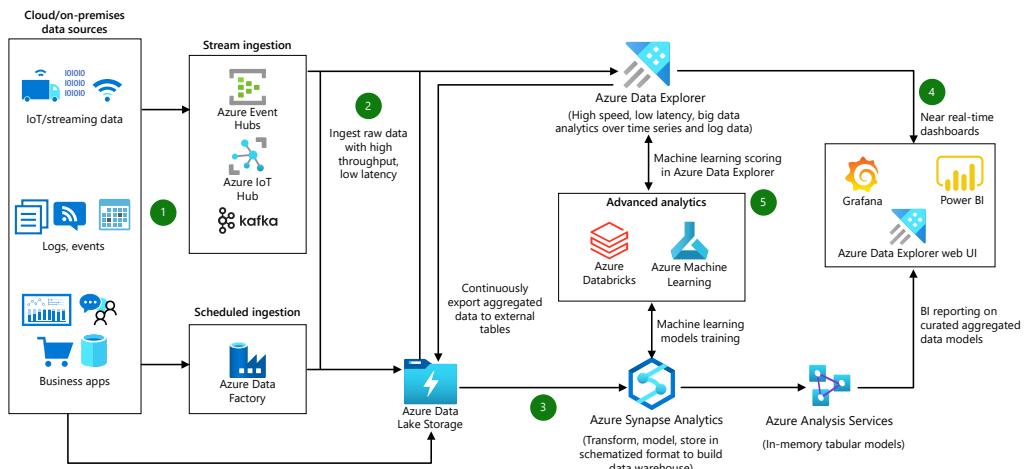
💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This solution idea demonstrates big data analytics over large volumes of high-velocity data from various sources.

Apache® and Apache Kafka® are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.

Architecture



Microsoft Azure



Download a [Visio file](#) of this architecture.

Dataflow

1. Raw structured, semi-structured, and unstructured (free text) data such as any type of logs, business events, and user activities can be ingested into Azure Data Explorer from various sources.
2. Ingest data into Azure Data Explorer with low-latency and high throughput using its connectors for [Azure Data Factory](#), [Azure Event Hubs](#), [Azure IoT Hub](#), [Kafka](#), and so on. Alternatively, ingest data through Azure Storage ([Blob](#) or [ADLS Gen2](#)), which uses [Azure Event Grid](#) and triggers the ingestion pipeline to Azure Data Explorer. You can also continuously export data to Azure Storage in compressed, partitioned parquet format and seamlessly query that data as detailed in the [Continuous data export overview](#).
3. Export pre-aggregated data from Azure Data Explorer to Azure Storage, and then ingest the data into Synapse Analytics to build data models and reports.
4. Use Azure Data Explorer's native capabilities to process, aggregate, and analyze data. To get insights at a lightning speed, build near real-time analytics dashboards using [Azure Data Explorer dashboards](#), [Power BI](#), [Grafana](#), or other tools. Use Azure Synapse Analytics to build a modern data warehouse and combine it with the Azure Data Explorer data to generate BI reports on curated and aggregated data models.
5. Azure Data Explorer provides native advanced analytics capabilities for [time series analysis](#), pattern recognition, [anomaly detection and forecasting](#), and [machine learning](#). Azure Data Explorer is also well integrated with ML services such as [Databricks](#) and [Azure Machine Learning](#). This integration allows you to build models using other tools and services and export ML models to Azure Data Explorer for scoring data.

Components

- [Azure Event Hubs](#) : Fully managed, real-time data ingestion service that's simple, trusted, and scalable.
- [Azure IoT Hub](#) : Managed service to enable bi-directional communication between IoT devices and Azure.
- [Kafka on HDInsight](#): Easy, cost-effective, enterprise-grade service for open source analytics with Apache Kafka.
- [Azure Data Explorer](#) : Fast, fully managed and highly scalable data analytics service for real-time analysis on large volumes of data streaming from applications, websites, IoT devices, and more.
- [Azure Data Explorer Dashboards](#): Natively export Kusto queries that were explored in the Web UI to optimized dashboards.
- [Azure Synapse Analytics](#) : Analytics service that brings together enterprise data warehousing and Big Data analytics.

Scenario details

Potential use cases

This solution illustrates how Azure Data Explorer and Azure Synapse Analytics complement each other for near real-time analytics and modern data warehousing use cases.

This solution is already being used by Microsoft customers. For example, the Singapore-based ride-hailing company, Grab, implemented real-time analytics over a huge amount of data collected from their taxi and food delivery services as well as merchant partner apps. The [team from Grab presented their solution at MS Ignite in this video \(20:30 onwards\)](#). Using this pattern, Grab processed more than a trillion events per day.

This solution is optimized for the retail industry.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Ornat Spodek](#) | Senior Content Manager

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Azure Data Explorer documentation](#)
- [Training: Introduction to Azure Data Explorer](#)
- [Azure Synapse Analytics](#)
- [Azure Event Hubs](#)

Related resources

- [Analytics architecture design](#)
- [Analytics end-to-end with Azure Synapse](#)
- [Advanced analytics architecture](#)

De-batch and filter serverless event processing with Event Hubs

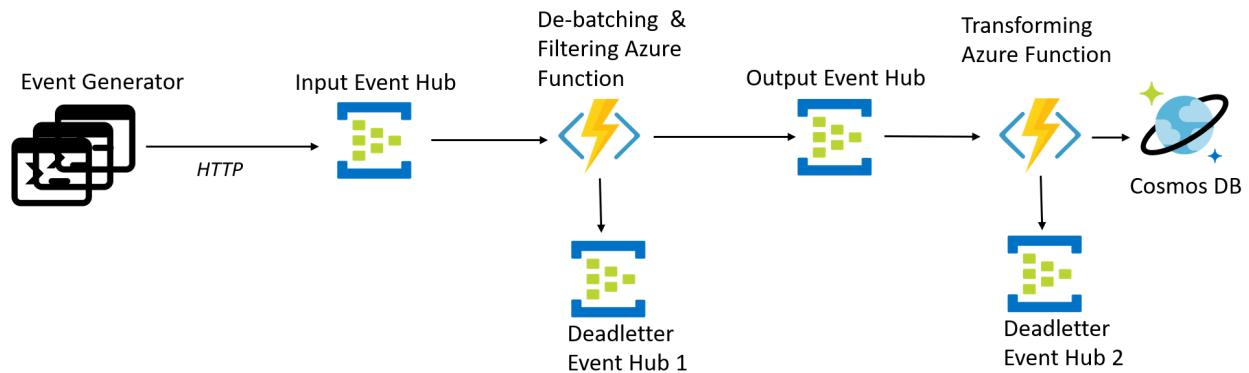
Azure Event Hubs Azure Functions Azure Cosmos DB

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes a serverless event-driven architecture that uses Azure Event Hubs and Azure Functions to ingest and filter a stream of data for database storage.

Architecture



Dataflow

1. Events arrive at the Input Event Hub.
2. The De-batching and Filtering Azure Function is triggered to handle the event. This step filters out unwanted events and de-batches the received events before submitting them to the Output Event Hub.
3. If the De-batching and Filtering Azure Function fails to store the event successfully, the event is submitted to the Deadletter Event Hub 1.
4. Events arriving at the Output Event Hub trigger the Transforming Azure Function. This Azure Function transforms the event into a message for the Azure Cosmos DB instance.
5. The event is stored in an Azure Cosmos DB database.

6. If the Transforming Azure Function fails to store the event successfully, the event is saved to the Deadletter Event Hub 2.

Components

- [Event Hubs](#) ingests the data stream. Event Hubs is designed for high-throughput data streaming scenarios.
- [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (a *function*) is invoked by a trigger.
- [Azure Cosmos DB](#) is a multi-model database service that is available in a serverless, consumption-based mode. For this scenario, the event-processing function stores JSON records, using the [Azure Cosmos DB for NoSQL](#).

Scenario details

This solution idea describes a variation of a serverless event-driven architecture that uses Event Hubs and Azure Functions to ingest and process a stream of data. The results are written to a database for storage and future review after they're de-batched and filtered.

To learn more about the basic concepts, considerations, and approaches for serverless event processing, consult the [Serverless event processing](#) reference architecture.

Potential use cases

A popular use case for implementing an end-to-end event stream processing pattern includes the Event Hubs streaming ingestion service to receive and process events per second using a de-batching and transformation logic implemented with highly scalable, event hub-triggered functions.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rajasa Savant](#) | Senior Software Development Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Azure Event Hubs documentation](#)
- [Introduction to Azure Functions](#)
- [Azure Functions documentation](#)
- [Overview of Azure Cosmos DB](#)
- [Choose an API in Azure Cosmos DB](#)

Related resources

- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [Monitoring serverless event processing](#) provides an overview and guidance on monitoring serverless event-driven architectures like this one.
- [Azure Kubernetes in event stream processing](#) describes a variation of a serverless event-driven architecture running on Azure Kubernetes with KEDA scaler.
- [Private link scenario in event stream processing](#) is a solution idea for implementing a similar architecture in a virtual network with private endpoints, in order to enhance security.

HIPAA and HITRUST compliant health data AI

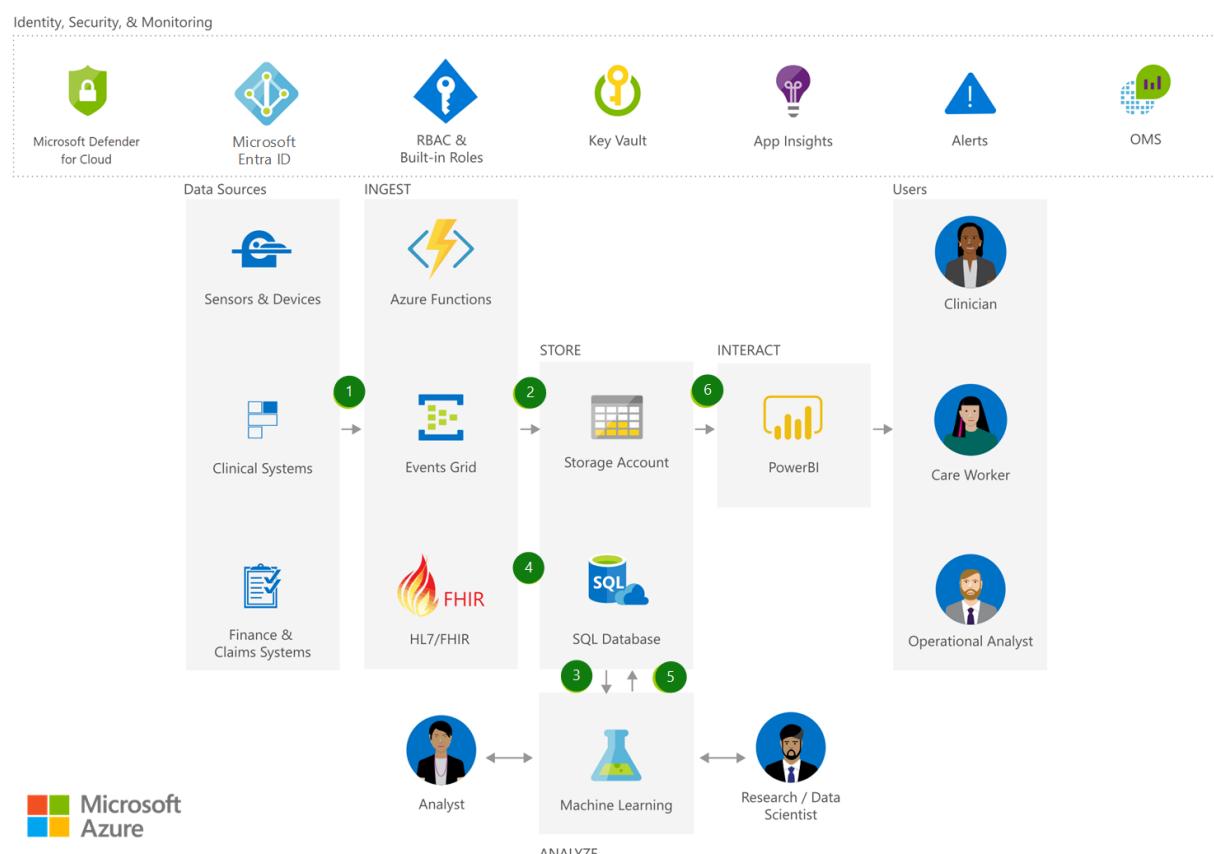
Azure Blob Storage Azure Event Grid Azure Functions Azure Machine Learning Power BI

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes how you can store, manage, and analyze HIPAA-compliant and HITRUST-compliant health data and medical records with a high level of built-in security.

Architecture



Download an [SVG](#) of this architecture.

Dataflow

1. Securely ingest bulk patient data into [Azure Blob storage](#).
2. [Event Grid](#) publishes patient data to [Azure Functions](#) for processing, and securely stores patient data in [SQL Database](#).
3. Analyze patient data using [Machine Learning](#), and create a Machine Learning-trained model.
4. Ingest new patient data in HL7/FHIR format and publish to [Azure Functions](#) for processing. Store in [SQL Database](#).
5. Analyze newly ingested data using the trained Machine Learning model.
6. Interact with patient data using [Power BI](#) while preserving Azure role-based access control (Azure RBAC).

Components

- [Azure Functions](#): Process events with serverless code
- [Event Grid](#): Get reliable event delivery at massive scale
- [Storage Accounts](#): Durable, highly available, and massively scalable cloud storage
- [Azure SQL Database](#): Managed, intelligent SQL in the cloud
- [Azure Machine Learning](#): Bring AI to everyone with an end-to-end, scalable, trusted platform with experimentation and model management
- [Power BI Embedded](#): Embed fully interactive, stunning data visualizations in your applications
- [Defender for Cloud](#): Unify security management and enable advanced threat protection across hybrid cloud workloads
- [Microsoft Entra ID](#): Synchronize on-premises directories and enable single sign-on
- [Key Vault](#): Safeguard and maintain control of keys and other secrets
- Application Insights: Detect, triage, and diagnose issues in your web apps and services
- [Azure Monitor](#): Full observability into your applications, infrastructure, and network
- [Operation Management Suite](#): A collection of management services that were designed in the cloud from the start
- [Azure RBAC and built-in roles](#): Azure role-based access control (Azure RBAC) has several built-in role definitions that you can assign to users, groups, and service principals.

Scenario details

This solution demonstrates how you can store, manage, and analyze HIPAA-compliant and HITRUST-compliant health data and medical records with a high level of built-in security.

Potential use cases

This solution is ideal for the medical and healthcare industry.

Next steps

- [Azure Functions Documentation](#)
- [Azure Event Grid Documentation](#)
- [Azure Storage Documentation](#)
- [Azure SQL Database Documentation](#)
- [Azure Machine Learning Documentation](#)
- [Power BI Embedded Documentation](#)
- [Microsoft Defender for Cloud Documentation](#)
- [Get started with Microsoft Entra ID](#)
- [What is Azure Key Vault?](#)
- [What is Application Insights?](#)
- [Monitoring Azure applications and resources](#)
- [What is Operations Management Suite \(OMS\)?](#)
- [Built-in roles for Azure role-based access control](#)

Related resources

- [Health data consortium on Azure](#)
- [Virtual health on Microsoft Cloud for Healthcare](#)
- [Confidential computing on a healthcare platform](#)

Serverless functions architecture design

Article • 07/28/2022

Serverless architecture evolves cloud platforms toward pure cloud-native code by abstracting code from the infrastructure that it needs to run. [Azure Functions](#) is a serverless compute option that supports *functions*, small pieces of code that do single things.

Benefits of using serverless architectures with Functions applications include:

- The Azure infrastructure automatically provides all the updated servers that applications need to keep running at scale.
- Compute resources allocate dynamically, and instantly autoscale to meet elastic demands. Serverless doesn't mean "no server," but "less server," because servers run only as needed.
- Micro-billing saves costs by charging only for the compute resources and duration the code uses to execute.
- Function *bindings* streamline integration by providing declarative access to a wide variety of Azure and third-party services.

Functions are *event-driven*. An external event like an HTTP web request, message, schedule, or change in data *triggers* the function code. A Functions application doesn't code the trigger, only the response to the trigger. With a lower barrier to entry, developers can focus on business logic, rather than writing code to handle infrastructure concerns like messaging.

Azure Functions is a managed service in Azure and Azure Stack. The open source Functions runtime works in many environments, including Kubernetes, Azure IoT Edge, on-premises, and other clouds.

Serverless and Functions require new ways of thinking and new approaches to building applications. They aren't the right solutions for every problem. For example serverless Functions scenarios, see [Reference architectures](#).

Implementation steps

Successful implementation of serverless technologies with Azure Functions requires the following actions:

- Decide and plan

Architects and technical decision makers (TDMs) perform [application assessment](#), conduct or attend [technical workshops and trainings](#), run [proof of concept \(PoC\)](#) or [pilot](#) projects, and conduct architectural designs sessions as necessary.

- [Develop and deploy apps](#)

Developers implement serverless Functions app development patterns and practices, configure DevOps pipelines, and employ site reliability engineering (SRE) best practices.

- [Manage operations](#)

IT professionals identify hosting configurations, future-proof scalability by automating infrastructure provisioning, and maintain availability by planning for business continuity and disaster recovery.

- [Secure apps](#)

Security professionals handle Azure Functions security essentials, secure the hosting setup, and provide application security guidance.

Related resources

- To learn more about serverless technology, see the [Azure serverless documentation](#).
- To learn more about Azure Functions, see the [Azure Functions documentation](#).
- For help with choosing a compute technology, see [Choose an Azure compute service for your application](#).

Serverless application architectures using Event Grid

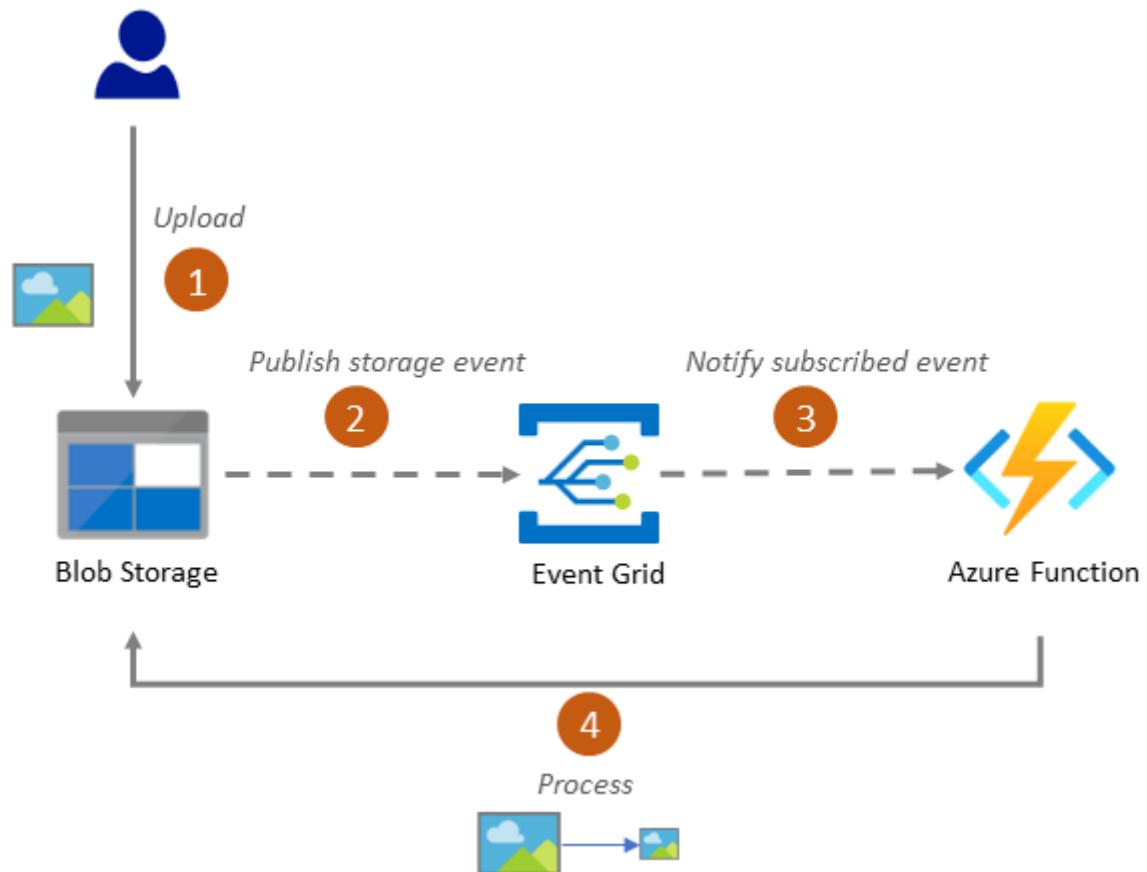
Azure Event Grid Azure Blob Storage

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes how to use Azure Event Grid to connect data sources and event handlers. The solution triggers a serverless function to run image analysis when a new photo enters an Azure Blob Storage container.

Architecture



Dataflow

1. A user uploads a photo to a Blob storage container.
2. Blob Storage publishes storage object events to Event Grid.
3. Event Grid triggers an Azure Function, based on the event criteria that the Function subscribed.
4. The function retrieves the photo and runs the image process on it (such as to shrink an image). Then it saves the new image to another Blob storage container.

Components

- [Azure Event Grid](#)
- [Azure Functions](#)
- [Azure Blob Storage](#)

Scenario details

The core design concept uses Event Grid to connect data sources and event handlers. Event Grid decouples event publishers from event subscribers by using a pub/sub model and a simple HTTP-based event delivery. This process allows the system to build scalable serverless applications.

Potential use cases

This solution idea publishes Blob Storage events by using Azure Event Grid. Then Azure Functions receives the event with built-in Event Grid support and processes the data in Blob Storage. Developers only need to focus on implementing the business logic in Azure Functions using this solution. Event Grid provides a reliable near-real-time notifications system for the event-driven integration between Blob Storage and Azure Functions. For example, Event Grid instantly triggers a serverless function to run an image process (such as to shrink an image), whenever someone adds a new photo to a Blob Storage container.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Herman Wu](#) | Senior Software Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Learn more about the component technologies:

- [What is Azure Event Grid?](#)
- [Introduction to Azure Functions](#)
- [Introduction to Azure Blob storage](#)

Related resources

Explore related architectures:

- [Application integration using Event Grid](#)
- [Event-based cloud automation](#)
- [Gridwich cloud media system](#)

Serverless apps using Azure Cosmos DB

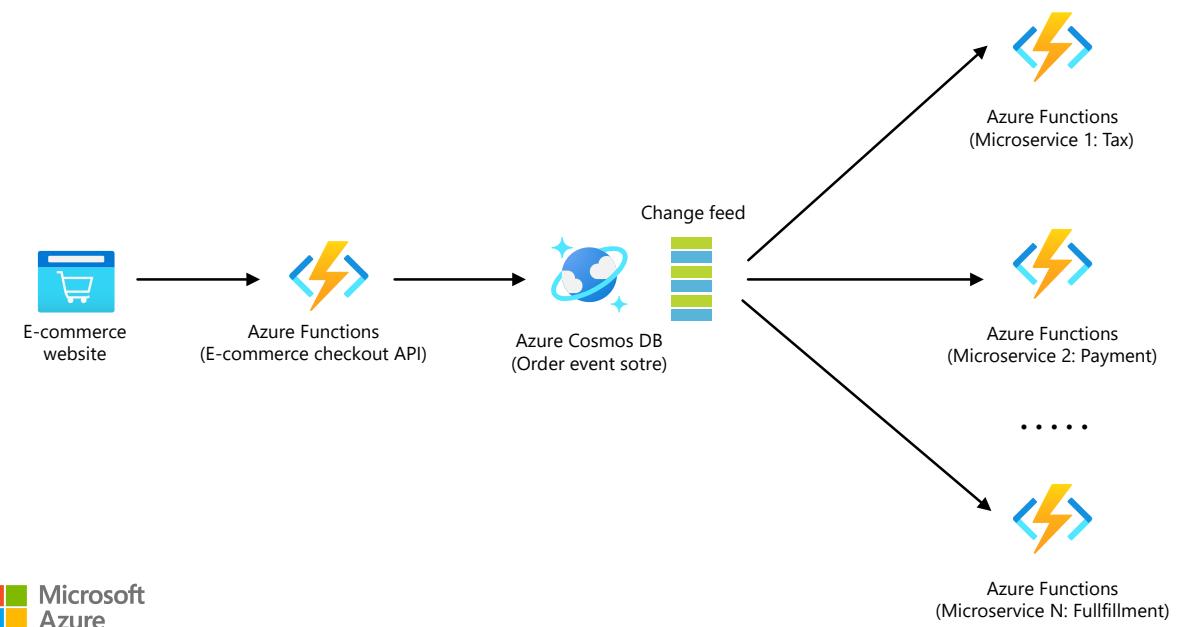
Azure Cosmos DB Azure Functions

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

You can use Azure Functions and Azure Cosmos DB to build globally distributed, scalable serverless applications.

Architecture



Download a [Visio file](#) of this architecture.

Dataflow

- A customer places an order in an e-commerce website.
- The order triggers an instance of Functions. The function processes the customer's checkout and stores information about the order in Azure Cosmos DB.

- The database insert operation triggers an Azure Cosmos DB change feed event.
- Systems that subscribe to change feed events are notified.
- The change feed notifications trigger Functions:
 - A function applies taxes to the order.
 - A function processes payment for the order.
 - A function fulfills the order.

Components

- [Functions](#) is an event-driven serverless compute platform. With Functions, you can use triggers and bindings to integrate services at scale.
- [Azure Cosmos DB](#) is a globally distributed, multi-model database. With Azure Cosmos DB, your solutions can elastically scale throughput and storage across any number of geographic regions.

Scenario details

Microservices offer many benefits:

- They provide highly scalable solutions.
- You can deploy each service independently.
- Fault isolation is straightforward when you confine functionality to separate containers.
- They fit well in a DevOps environment.
- They decrease time to market by speeding up the software development lifecycle.

An efficient way to implement microservices is to use a serverless technology. This solution uses Functions, an Azure offering that provides a serverless compute experience. The solution uses Azure Cosmos DB for data storage. Azure Cosmos DB offers a change feed that integrates with Functions.

Potential use cases

This solution applies to many areas:

- E-commerce
- Retail
- Inventory management

Next steps

- [Introduction to Azure Functions](#)
- [Welcome to Azure Cosmos DB](#)
- [Change feed in Azure Cosmos DB](#)
- [Create a function triggered by Azure Cosmos DB](#)
- [Connect Azure Functions to Azure Cosmos DB using Visual Studio Code](#)

Related resources

See the following architectures that include Functions and Azure Cosmos DB:

- [Azure Cosmos DB in IoT workloads](#)
- [Transactional Outbox pattern with Azure Cosmos DB](#)
- [Gaming using Azure Cosmos DB](#)
- [Code walkthrough: Serverless application with Functions](#)
- [Analyze news feeds with near real-time analytics using image and natural language processing](#)

See the following architectures that feature Functions:

- [Integrate Event Hubs with serverless functions on Azure](#)
- [Azure Functions in a hybrid environment](#)
- [Monitor Azure Functions and Event Hubs](#)
- [Azure App Service and Azure Functions considerations for multitenancy](#)
- [Performance and scale for Event Hubs and Azure Functions](#)

See the following architectures that feature Azure Cosmos DB:

- [Visual search in retail with Azure Cosmos DB](#)
- [Scalable order processing](#)
- [Deliver highly scalable customer service and ERP applications](#)
- [CI/CD pipeline for container-based workloads](#)

Serverless computing solution for LOB apps

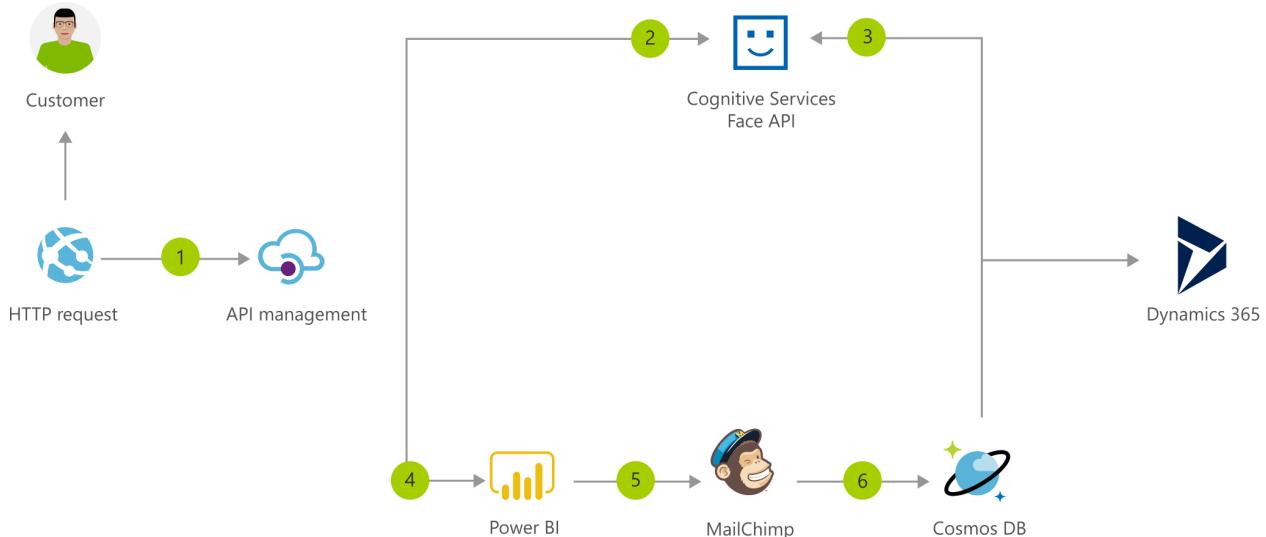
Azure AI services Azure Cosmos DB Dynamics 365 Power BI

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This serverless solution provides an efficient way to manage customer data. Core components include the Azure Cognitive Services Face API, which offers access to facial recognition technology. The solution also includes customer relationship management (CRM) through Dynamics 365 and data analytics through Power BI.

Architecture



[Download an SVG of this architecture.](#)

Dataflow

1. Information about a new customer is posted to a web endpoint.
2. The customer's photo is posted to the [Cognitive Services Face API](#), where the image is linked to the customer's name.
3. The customer information is recorded in a CRM system such as [Dynamics 365](#).

4. The customer information is sent to [Power BI](#).
5. The customer information is added to a MailChimp mailing list.
6. The solution creates a record of the customer in [Azure Cosmos DB](#).

Components

- [Azure API Management](#) creates consistent, modern API gateways for back-end services. Besides accepting API calls and routing them to back ends, this platform also verifies keys, tokens, certificates, and other credentials. API Management also enforces usage quotas and rate limits and logs call metadata.
- [Cognitive Services](#) consists of cloud-based services that provide AI functionality. You can use the REST APIs and client library SDKs to build cognitive intelligence into apps.
- The [Cognitive Services Face API](#) provides access to functionality that detects facial features and attributes. You can also use the API to match images.
- [Dynamics 365](#) is a portfolio of intelligent applications that businesses can use for enterprise resource planning (ERP) and CRM.
- [Power BI](#) is a collection of software services and apps that provide analytics reporting.
- [Mailchimp](#) is an email marketing platform that provides automation services.
- [Azure Cosmos DB](#) is a globally distributed, multi-model database. With Azure Cosmos DB, your solutions can elastically scale throughput and storage across any number of geographic regions.
- [Azure Functions](#) is a serverless compute platform that you can use to build applications. With Functions, you can use triggers and bindings to react to changes in Azure services.

Scenario details

Serverless architectures, like the one in this solution, offer many benefits. You can build and run applications without having to manage or maintain the underlying infrastructure. As a result, you can dramatically improve developer productivity.

This solution uses a NoSQL database, Azure Cosmos DB. This type of database system is designed to quickly store huge volumes of rapidly changing, unstructured data and make it readily available for search, consolidation, and analysis.

Potential use cases

This solution benefits organizations that manage large volumes of customer data. It's ideal for retail, media and entertainment, and other industries that use service-based subscriptions to stream videos and applications like Office 365 and Adobe.

Next Steps

- [Learn to build Serverless apps](#)
- [What are Azure Cognitive Services?](#)
- [What is the Azure Face service?](#)
- [What is Azure API Management?](#)
- [What is Power BI?](#)
- [Welcome to Azure Cosmos DB](#)

Related resources

- [Decide which compute option to use for your apps](#)
- [Face recognition and sentiment analysis](#)

Serverless event stream processing in a VNet with private endpoints

Azure Private Link

Azure Event Hubs

Azure Functions

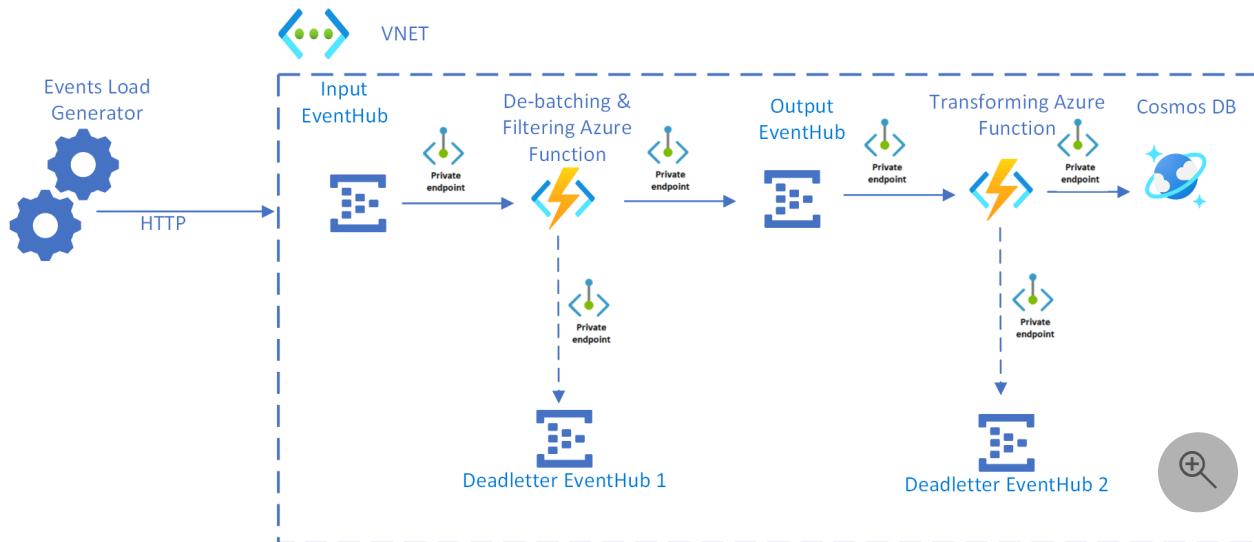
Azure Cosmos DB

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes a serverless event-driven architecture in a virtual network that ingests and processes a stream of data and then writes the results to a database.

Architecture



Dataflow

1. VNet integration is used to put all Azure resources behind [Azure Private Endpoints](#).
2. Events arrive at the Input Event Hub.
3. The De-batching and Filtering Azure Function is triggered to handle the event. This step filters out unwanted events and de-batches the received events before submitting them to the Output Event Hub.

4. If the De-batching and Filtering Azure Function fails to store the event successfully, the event is submitted to the Deadletter Event Hub 1.
5. Events arriving at the Output Event Hub trigger the Transforming Azure Function. This Azure Function transforms the event into a message for the Azure Cosmos DB instance.
6. The event is stored in an Azure Cosmos DB database.
7. If the Transforming Azure Function fails to store the event successfully, the event is saved to the Deadletter Event Hub 2.

 **Note**

For simplicity, subnets are not shown in the diagram.

Components

- [Azure Private Endpoint](#) is a network interface that connects you privately and securely to a service powered by Azure Private Link. Private Endpoint uses a private IP address from your VNet, effectively bringing the service into your VNet.
- [Event Hubs](#) ingests the data stream. Event Hubs is designed for high-throughput data streaming scenarios.
- [Azure Functions](#) is a serverless compute option. It uses an event-driven model, where a piece of code (a *function*) is invoked by a trigger.
- [Azure Cosmos DB](#) is a multi-model database service that is available in a serverless, consumption-based mode. For this scenario, the event-processing function stores JSON records, using the [Azure Cosmos DB for NoSQL](#).

Scenario details

This solution idea shows a variation of a serverless event-driven architecture that ingests a stream of data, processes the data, and writes the results to a back-end database. In this example, the solution is hosted inside a virtual network with all Azure resources behind private endpoints.

To learn more about the basic concepts, considerations, and approaches for serverless event processing, consult the [Serverless event processing](#) reference architecture.

Potential use cases

A popular use case for implementing an end-to-end event stream processing pattern includes the Event Hubs streaming ingestion service to receive and process events per

second by using de-batching and transformation logic implemented with highly scalable functions triggered by Event Hubs.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rajasa Savant](#) | Senior Software Development Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Manage a Private Endpoint connection](#)
- [Private Endpoint quickstart guides:](#)
 - [Create a Private Endpoint using the Azure portal](#)
 - [Create an Azure Private Endpoint using Azure PowerShell](#)
 - [Create a Private Endpoint using Azure CLI](#)
 - [Create a Private Endpoint by using an ARM template](#)
- [Azure Event Hubs documentation](#)
- [Introduction to Azure Functions](#)
- [Azure Functions documentation](#)
- [Overview of Azure Cosmos DB](#)
- [Choose an API in Azure Cosmos DB](#)

Related resources

- [Serverless event processing](#) is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- [Monitoring serverless event processing](#) provides an overview and guidance on monitoring serverless event-driven architectures like this one.
- [De-batching and filtering in serverless event processing with Event Hubs](#) describes in more detail how these portions of the architecture work.
- [Azure Kubernetes in event stream processing](#) describes a variation of a serverless event-driven architecture running on Azure Kubernetes with KEDA scaler.

Transit hub dynamic pub-sub messaging system

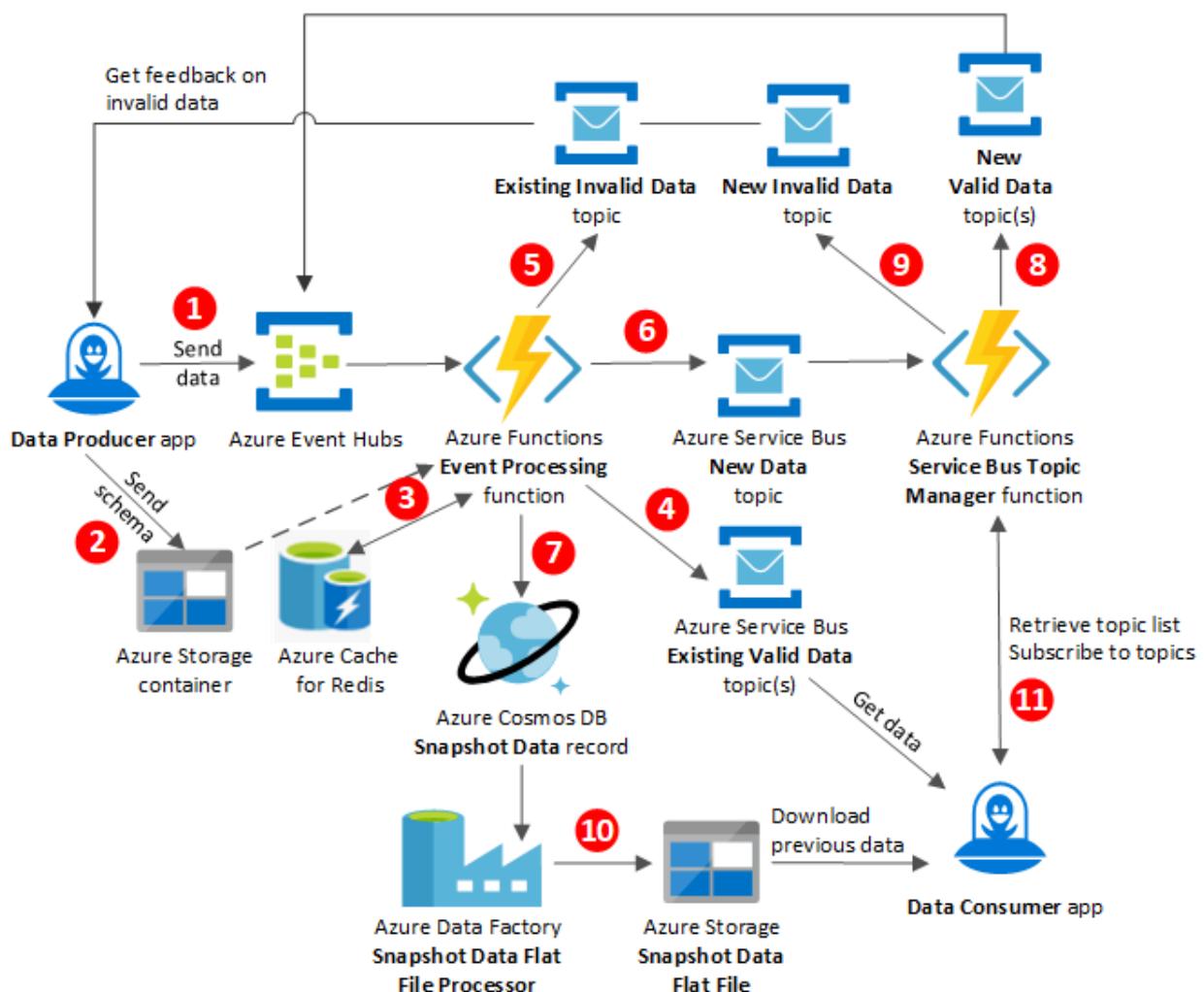
Azure Cache for Redis Azure Cosmos DB Azure Event Hubs Azure Functions Azure Service Bus

💡 Solution ideas

This article is a solution idea. If you'd like us to expand the content with more information, such as potential use cases, alternative services, implementation considerations, or pricing guidance, let us know by providing [GitHub feedback](#).

This article describes an elastic, flexible [publish-subscribe model](#) for data producers and consumers to create and consume validated, curated content or data.

Architecture



Download a [Visio file](#) of this architecture.

Dataflow

1. The **Data Producer** app publishes data to Azure Event Hubs, which sends the data to the Azure Functions **Event Processing** function.
2. The **Data Producer** also sends the JSON schema for storage in an Azure Storage container.
3. The **Event Processing** function retrieves the JSON schema from Azure Cache for Redis to reduce latency, and uses the schema to validate the data.

If the schema isn't cached yet, the **Event Processing** function retrieves the schema from the Azure Storage container. The request for the schema also stores the schema in Azure Cache for Redis for future retrieval.

 **Note**

Azure Schema Registry in Event Hubs can be a viable alternative to storing and caching JSON schemas. For more information, see [Azure Schema Registry in Event Hubs \(Preview\)](#).

4. If a topic already exists and the data is valid, the **Event Processing** function merges the data into the existing **Valid Data** Azure Service Bus topic, and sends the topic to the **Data Consumer** app.
5. If a topic already exists and the data is invalid, the **Event Processing** function merges the data into the existing **Invalid Data** Service Bus topic, and sends the topic back to the data producer. The data producer subscribes to the **Invalid Data** topics to get feedback about invalid data that the producer created.
6. If a topic doesn't exist yet, the **Event Processing** function publishes the new data to a **New Data** Service Bus topic, and sends the topic to the **Service Bus Topic Manager** function.
7. If the new data is valid, the **Event Processing** function also inserts the data as a new **Snapshot Data** record in Azure Cosmos DB.
8. If the new data is valid, the **Service Bus Topic Manager** function creates a new **Valid Data** Service Bus topic, and sends the topic to Event Hubs.

9. If the new data is invalid, the **Service Bus Topic Manager** function creates a new **Invalid Data** Service Bus topic, and sends the topic back to the **Data Producer** app.
10. The **Snapshot Data Flat File Processor** in Azure Data Factory runs on a schedule to extract all snapshot data from the **Snapshot Data** Azure Cosmos DB database. The processor creates a flat file and publishes it to a **Snapshot Data Flat File** in Azure Storage for downloads.
11. The **Data Consumer** app retrieves a list of all the Service Bus topics that the **Service Bus Topic Manager** has available for subscription. The app registers with the **Service Bus Topic Manager** to subscribe to Service Bus topics.

Components

- [Azure Event Hubs](#) ↗
- [Azure Service Bus](#) ↗
- [Azure Functions](#) ↗
- [Azure Data Factory](#) ↗
- [Azure Cosmos DB](#) ↗
- [Azure Blob Storage](#) ↗
- [Azure Cache for Redis](#) ↗

Scenario details

The Transit Hub is a dynamic [publish-subscribe model](#) for data producers and data consumers to create and consume validated, curated content or data. The model is elastic to allow for scale and performance. Data producers can quickly onboard and upload data to a service. The service validates the data against a schema that the data producer provides. The service then makes the validated data available for subscribers to consume data they're interested in.

The service validating the data doesn't need to know about the payload, only whether it's valid against the schema that the producer provides. This flexibility means the service can accept new payload types without having to be redeployed. This solution also lets data consumers get historical data that was published before the consumer subscribed.

Potential use cases

This model is especially useful in the following scenarios:

- Messaging systems where user volume and status are unknown or vary unpredictably
- Publishing systems that potentially need to support new or unknown data sources
- Commerce or ticketing systems that need to continually update data and cache it for fast delivery

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- [Rick Weyenberg](#) ↗ | Principal Cloud Solution Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Azure Web PubSub service documentation](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Tutorial: Create a serverless notification app with Azure Functions and Azure Web PubSub service](#)

Related resources

- [Publisher-Subscriber pattern](#)
- [Asynchronous messaging options in Azure](#)

DevOps architecture design

Article • 07/05/2023

The term *DevOps* derives from *development* and *operations*. It refers to the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. For an overview of DevOps, see [What is DevOps?](#).

DevOps includes these activities and operations:

- **Continuous integration (CI)** is the practice of merging all developer code into a central codebase frequently, and then performing automated build and test processes. The objectives are to quickly discover and correct code issues, to streamline deployment, and to ensure code quality. For more information, see [What is Continuous Integration?](#).
- **Continuous delivery (CD)** is the practice of automatically building, testing, and deploying code to production-like environments. The objective is to ensure that code is always ready to deploy. Adding continuous delivery to create a full CI/CD pipeline helps you detect code defects as soon as possible. It also ensures that properly tested updates can be released in a short time. For more information, see [What is Continuous Delivery?](#).
- **Continuous deployment** is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning. It might not be appropriate for all teams.
- **Continuous monitoring** refers to the process and technology required to incorporate monitoring across each phase of DevOps and IT operations lifecycles. Monitoring helps to ensure the health, performance, and reliability of your application and infrastructure as the application moves from development to production. Continuous monitoring builds on the concepts of CI and CD.

Introduction to DevOps on Azure

If you need to know more about DevOps, or DevOps on Azure, the best place to learn is [Microsoft Learn training](#). This free online platform provides interactive training for Microsoft products and more. There are videos, tutorials, and hands-on learning for specific products and services, plus learning paths based on job role, such as developer or data analyst. If you're not familiar with Learn you can take [a tour of Microsoft Learn training](#) or [a quick video tour of Microsoft Learn training](#).

After you're familiar with Azure, you can decide whether to follow learning paths specific to DevOps, such as:

- Get started with Azure DevOps
- Deploy applications with Azure DevOps
- Build applications with Azure DevOps

[Browse other training materials for DevOps](#)

Path to production

Plan your path to production by reviewing:

- [DevOps guides](#)
- [Azure services that are often used in implementing DevOps solutions](#)
- [Example DevOps architectures](#)

DevOps guides

Article or section	Description
DevOps checklist	A list of things to consider and do when you implement DevOps attitudes and methods in culture, development, testing, release, monitoring, and management.
Operational Excellence patterns	A list of design patterns for achieving Operational Excellence—one of the five pillars of the Microsoft Azure Well-Architected Framework —in a cloud environment. See Cloud Design Patterns for more patterns.
Advanced Azure Resource Manager template functionality	Some advanced examples of template use.
DevTest Labs guidance	A series of articles to help you use Azure Devtest Labs to provision development and test environments. The first article in the series is DevTest Labs in the enterprise .
Azure Monitor guidance	A series of articles to help you use Azure Monitor to monitor cloud environments. The first article in the series is Azure Monitor best practices - Planning your monitoring strategy and configuration .
Continuous integration and delivery for an Azure Synapse Analytics workspace	An outline of how to use an Azure DevOps release pipeline and GitHub Actions to automate the deployment of an Azure Synapse workspace to multiple environments.

Article or section	Description
DevOps for quantum computing	A discussion of the DevOps requirements for hybrid quantum applications.
Platform automation for Azure VMware Solution enterprise-scale scenario	An overview for deploying Azure VMware Solution, including guidance for operational automation.

Azure DevOps services

Azure service	Documentation	Description
Azure Artifacts	Azure Artifacts overview	Fully integrated package management for your CI/CD pipelines.
Azure DevOps	Azure DevOps documentation	Modern dev services for managing your development lifecycle end-to-end. It includes Azure Repos, Azure Pipelines, and Azure Artifacts.
Azure DevTest Labs	Azure DevTest Labs documentation	Reusable templates and artifacts for provisioning development and test environments.
Azure Lab Services	Azure Lab Services documentation	A tool for setting up and providing on-demand access to preconfigured virtual machines (VMs).
Azure Monitor	Azure Monitor documentation	Provides full observability into your applications, infrastructure, and network.
Azure Pipelines	Azure Pipelines documentation	Helps you automate build and deployment by using cloud-hosted pipelines.
Azure Repos	Azure Repos documentation	Provides unlimited, cloud-hosted private Git repos for your project and can be configured to use GitHub Advanced Security.
Azure Resource Manager	Azure Resource Manager documentation	Provides consistent deployment, organization, and control for resource management.
Azure Resource Manager templates (ARM templates)	ARM template documentation	Templates that you can use to define the infrastructure and configuration for your project.
Azure Test Plans	Azure Test Plans documentation	Provides planned and exploratory testing services for your apps.

Example DevOps architectures

The DevOps architectures are found in two sections:

Section	First article in the section
Architectures	Automate multistage DevOps pipelines with Azure Pipelines
Solution ideas	CI/CD for Azure VMs

Here are some example architectures. For each one there's a list of the key Azure services used in the architecture.

Architecture	Description	Azure services used
Automate multistage DevOps pipelines with Azure Pipelines	Use Azure DevOps REST APIs to build CI/CD pipelines.	Azure DevOps, Logic Apps, Azure Pipelines
Automated API deployments with APIOps	Apply GitOps and DevOps techniques to ensure quality APIs.	Azure Repos, API Management, Azure DevOps, Azure Pipelines, Azure Repos
Design a CI/CD pipeline using Azure DevOps	Build a CI/CD pipeline by using Azure DevOps and other services.	Azure Repos, Azure Test Plans, Azure Pipelines
Teacher-provisioned virtual labs in Azure	Teachers can easily set up virtual machines for students to work on class exercises.	Lab Services
Enterprise monitoring with Azure Monitor	Use Azure Monitor to achieve enterprise-level monitoring and centralized monitoring management.	Azure Monitor

Best practices

The [Microsoft Azure Well-Architected Framework](#) provides reference guidance and best practices that you can use to improve the quality of your architectures. The framework comprises five pillars: Reliability, Security, Cost Optimization, Operational Excellence, and Performance Efficiency. Here's where to find documentation of the pillars:

- [Reliability](#)
- [Security](#)
- [Cost Optimization](#)
- [Operational Excellence](#)

- [Performance Efficiency](#)

The following articles are about best practices that are specific to DevOps and to some DevOps services.

DevOps

- [How Teams at Microsoft Embraced a DevOps Culture - Azure webinar series ↗](#)
- [DevOps checklist](#)
- [Azure cloud migration best practices checklist](#)
- [Resiliency checklist for specific Azure services](#)
- [Continuous monitoring with Azure Monitor](#)
- [Monitoring best practices for reliability in Azure applications](#)
- [Overview of the Azure Security Benchmark \(v1\)](#)
- [Azure Identity Management and access control security best practices](#)
- [Security best practices](#)
- [Azure security best practices and patterns](#)
- [Azure operational security checklist](#)
- [Azure security baseline for API Management](#)
- [Secure development best practices on Azure](#)

Azure Artifacts

- [Azure Artifacts: best practices](#)

Azure Resource Manager

- [ARM template best practices](#)
- [Best practices for Bicep](#)

Stay current with DevOps

Stay current with Azure DevOps by monitoring these articles:

- [Azure DevOps Feature Timeline](#)
- [Azure DevOps documentation - what's new?](#)

Additional resources

Example solutions

- Design a CI/CD pipeline using Azure DevOps
- Manage Microsoft 365 tenant configuration by using Microsoft365DSC and Azure DevOps
- Run containers in a hybrid environment

AWS or Google Cloud professionals

- AWS to Azure services comparison - DevOps and application monitoring
- Google Cloud to Azure services comparison - DevOps and application monitoring

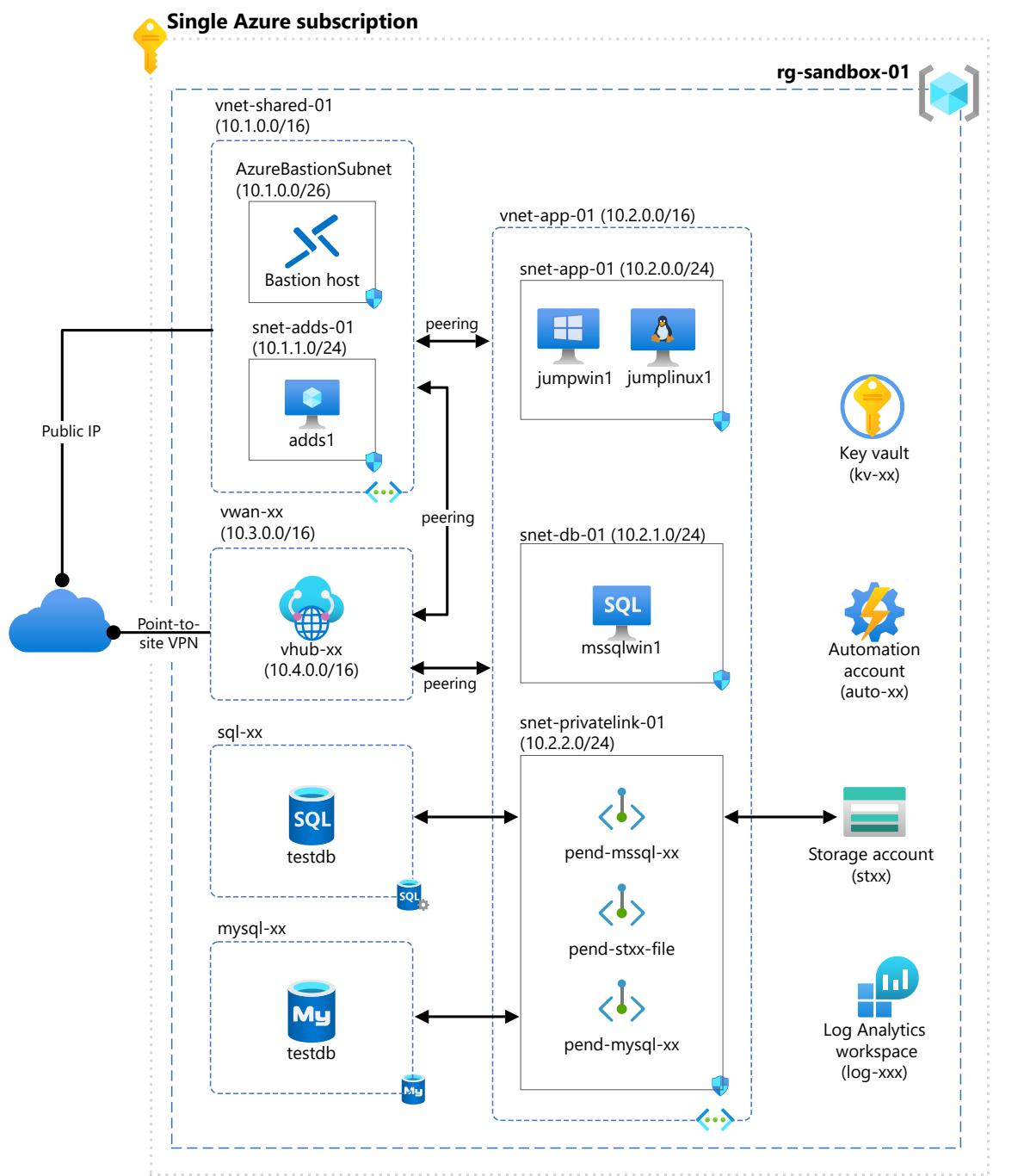
Azure Sandbox

Azure Bastion Azure Database for MySQL Azure SQL Database SQL Server on Azure Virtual Machines
Azure Virtual Machines

Azure Sandbox is a collection of interdependent [cloud computing](#) configurations for implementing common [Azure](#) services on a single [subscription](#). This collection provides a flexible and cost effective sandbox environment for experimenting with Azure services and capabilities.

Depending on your Azure offer type and region, a fully provisioned Azure Sandbox environment can be expensive to run. You can reduce costs by stopping or deallocating virtual machines (VMs) when not in use, or by skipping optional configurations that you don't plan to use.

Architecture



Download a [Visio file](#) of this architecture.

Components

You can deploy all the following sandbox configurations, or just the ones you need.

- Shared services virtual network, [Azure Bastion](#), and Active Directory domain controller

- Application virtual network, Windows Server jump box, Linux jump box, and [Azure Files](#) share
- [SQL Server on Azure Virtual Machines](#)
- [Azure SQL Database](#)
- [Azure Database for MySQL Flexible Server](#)
- [Azure Virtual WAN](#) and point-to-site VPN

Deploy the sandbox

The Azure Sandbox environment requires the following prerequisites:

- A [Microsoft Entra ID](#) tenant
- An [Azure subscription](#)
- The appropriate [Azure role-based access control \(RBAC\)](#) role assignments
- A [service principal](#)
- A [configured client environment](#)

For more information about how to prepare for a sandbox deployment, see [Prerequisites](#).

To integrate [AzureSandbox](#) with an [Azure landing zone](#) consider doing the following:

- Place the sandbox subscription in the *Sandboxes* management group.
- Keep the sandbox isolated from your private network.
- Audit sandbox subscription activity.
- Limit sandbox access, and remove access when it is no longer required.
- Decommission sandboxes after an expiration period to control costs.
- Create a budget on sandbox subscriptions to control costs.

See [Landing zone sandbox environments](#) for more information.

To deploy Azure Sandbox, go to the [AzureSandbox](#) GitHub repository and begin with [Getting started](#). See [Default Sandbox Deployment](#) to deploy your Azure Sandbox environment. For more information, see [Known issues](#).

Use cases

A sandbox is ideal for accelerating Azure projects. After you deploy your sandbox environment, you can add services and capabilities. You can use the sandbox for activities like:

- Self-learning

- Hackathons
- Testing
- Development

Important

Azure Sandbox isn't intended for production use. The deployment uses some best practices, but others intentionally aren't used in favor of simplicity and cost.

Capabilities

Have you ever wanted to experiment with a particular Azure service or capability, but were blocked by all the foundational prerequisites? A sandbox environment can accelerate your project by provisioning many of the mundane core infrastructure components. You can focus on just the services or capabilities you need to work with.

For example, you can use the following capabilities and configurations that the Azure Sandbox environment provides:

- Connect to a Windows jump box VM from the internet.
 - Option 1: Internet-facing access by using a web browser and Azure Bastion
 - Option 2: Point-to-site VPN connectivity via Azure Virtual WAN
- Use a preconfigured Active Directory Domain Services local domain as a domain administrator.
 - Preconfigured integrated DNS server
 - Preconfigured integration with Azure private DNS zones
 - Preconfigured integration with Azure Private Link private endpoints.
- Use an Azure Files preconfigured file share.
- Use a Windows jumpbox VM as a developer workstation.
 - Domain joined to local domain
 - Administer Active Directory and DNS with preinstalled Windows Server Remote Server Administration Tools
 - Visual Studio Code preinstalled with Remote-SSH into a Linux jump box
 - Azure Storage Explorer, AzCopy, and Azure Data Studio preinstalled
 - SQL Server Management Studio preinstalled
 - MySQL Workbench preinstalled
- Use a Linux jump box VM as a DevOps agent.
 - Domain joined to local domain using Winbind

- Azure CLI, PowerShell, and Terraform preinstalled
- Dynamic CIFS mount to Azure Files preconfigured file share
- Use a preconfigured SQL Server VM.
 - Domain joined to local domain
- Use a preconfigured Azure SQL database or Azure Database for MySQL Flexible Server through private endpoints.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributor.

Principal author:

- [Roger Doherty ↗](#)

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- [Develop and test on Azure ↗](#)
- [Microsoft Cloud Adoption Framework](#)
- [Cloud Adoption Framework Azure setup guide](#)
- [Microsoft Azure Well-Architected Framework](#)

Related resources

- [Technology choices for Azure solutions](#)
- [Best practices for cloud applications](#)
- [Build applications on the Microsoft Cloud](#)

DevOps checklist

Article • 03/13/2023

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

Culture

Ensure business alignment across organizations and teams. Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that business, development, and operations teams are aligned.

Ensure that your team understands your software life cycle. Your team needs to understand the overall life cycle of your applications, and where each application is within that life cycle. Having this information helps all team members know what they should do now, and what they should plan and prepare for in the future.

Reduce cycle time. Aim to minimize the time that it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations teams.

Review and improve processes. Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

Do proactive planning. Proactively plan for failure. Have processes in place to quickly identify problems when they occur, escalate problems to the correct team members to fix, and confirm their resolution.

Learn from failures. Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the problem, document the cause and solution, and share any lessons that you learn. Whenever possible, update your build processes to automatically detect such failures in the future.

Optimize for speed, and collect data. Every planned improvement is a hypothesis. Work in the smallest increments that are possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess experiment effectiveness. Be ready to fail fast if the hypothesis is wrong.

Allow time for learning. Failures and successes provide opportunities for learning. Before you move on to new projects, allow time to gather important lessons, and make sure that your team absorbs those lessons. Also give your team time to build skills, experiment, and learn about new tools and techniques.

Document operations. Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems that you support, along with recovery processes and other maintenance procedures. Focus on the steps that you actually do, not theoretically optimal processes. Regularly review and update your documentation. For code, make sure to include meaningful comments, especially in public APIs. Use tools to generate code documentation automatically whenever possible.

Share knowledge. Documentation is only useful if people know that it exists and can find it. Keep your documentation organized, and make it easily discoverable. Be creative: use brown bags (informal presentations), videos, or newsletters to share knowledge.

Development

Provide developers with production-like environments. If development and test environments don't match your production environment, it's hard to test and diagnose problems. Keep development and test environments as close to your production environment as possible. Make sure that test data is consistent with the data that you use in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Ensure that all authorized team members can provision infrastructure and deploy applications. Setting up production-like resources and deploying an application shouldn't involve complicated manual tasks or detailed technical knowledge of a system. Anyone with the right permissions should be able to create or deploy production-like resources without going to your operations team.

This recommendation doesn't imply that anyone can push live updates to a production deployment. It's about reducing friction for development and QA teams to create production-like environments.

Instrument each application for insight. To understand the health of your applications, you need to know how they perform and whether they experience any errors or problems. Always include instrumentation as a design requirement, and build instrumentation into each application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the health and usage of each application.

Track your technical debt. Many projects prioritize release schedules over code quality to one degree or another. Always document when shortcuts are taken or other suboptimal implementations, and schedule time to revisit these issues.

Consider pushing updates directly to production. To reduce your overall release cycle time, consider pushing properly tested code commits directly to production. Use [feature toggles](#) to control which features you enable. Then you can move quickly from development to release by using the toggles to enable or disable features. Toggles are also useful when you perform tests like [canary releases](#), where you deploy a particular feature to a subset of your production environment.

Testing

Automate testing. Manually testing software is tedious and susceptible to error. Automate common testing tasks, and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. When you run integrated UI tests, also use an automated tool. Azure offers development and test resources that can help you configure and run testing. For more information, see [Develop and test on Azure](#).

Test for failures. When a system can't connect to a service, the system should respond gracefully. And when the service is available again, the system should recover. Make fault-injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

Test in production. A release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that you update infrequently, schedule production testing as a regular part of maintenance.

Automate performance testing to identify performance problems early. The impact of a serious performance problem can be as severe as a bug in code. Although automated functional tests can prevent application bugs, these tests might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline to make sure that your application meets those goals.

Perform capacity testing. An application might work fine under test conditions and then have problems in production because of scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure that the application can handle those limits, but also test what happens when you exceed those limits. Do capacity testing at regular intervals.

After an initial release, you should run performance and capacity tests whenever you update production code. Use historical data to fine-tune tests and to determine what types of tests you need to do.

Perform automated security penetration testing. Ensuring the security of your application is as important as testing any other functionality. Make automated penetration testing a standard part of your build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing doesn't remove the need for in-depth security reviews at regular intervals.

Perform automated business continuity testing. Develop tests for large-scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Release

Automate deployments. Automation provides many benefits, including:

- Enabling faster and more reliable deployments.
- Ensuring consistent deployments to any supported environment, including test, staging, and production.
- Removing the risk of human error that manual deployments can introduce.
- Making it easy to schedule releases for convenient times, which minimizes any effects of potential downtime.

Automate the process of deploying each application to your test, staging, and production environments. Have systems in place to detect any problems during rollout, and have an automated way to roll forward fixes or roll back changes.

Use continuous integration. Continuous integration (CI) is the practice of merging all developer code into a central code base on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a code base at the same time without conflicts. CI also helps you find code defects as early as possible. Preferably, a CI process should run every time that you commit or check in code. It should run at least once per day.

Consider adopting a [trunk-based development model](#). In this model, developers commit to a single branch (the trunk). There's a requirement that commits never break a build. This model facilitates CI, because you do all feature work in the trunk, and you resolve any merge conflicts when each commit happens.

Consider using continuous delivery. Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding CD to create a full CI/CD pipeline helps you detect code defects as soon as possible. It also ensures that you can release properly tested updates in a short time.

Continuous *deployment* is a process that automatically takes any updates that have passed through a CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning. It might not be appropriate for all teams.

Make small, incremental changes. Large code changes have a greater potential to introduce bugs than smaller ones do. Whenever possible, keep changes small. Doing so limits the potential effects of each change and simplifies the task of understanding and debugging problems.

Control exposure to changes. Make sure that you're in control of when updates become visible to your end users. Consider using feature toggles to control when you turn on features for end users.

Implement release management strategies to reduce deployment risk. Deploying an application update to production always entails some risk. To minimize this risk, use strategies like [canary releases](#) or [blue/green deployments](#) to deploy updates to a subset of users. Confirm that each update works as expected, and then roll out each update to the rest of the system.

Document all changes. Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches that you applied, policy changes, and configuration changes. The record of the changes should be visible to your entire team. But don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.

Consider making infrastructure immutable. Immutable infrastructure is based on the principle that you shouldn't modify infrastructure after you deploy it to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. With this approach, you can test and deploy your code and your hosting environment as a block. After deployment, you don't modify infrastructure components until the next build and deploy cycle.

Monitoring

Make systems observable. Your operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and code applications to instrument operations metrics. Use a common and consistent schema that helps you correlate events across systems. The standard method of tracking the health and status of Azure resources is to use [Azure Diagnostics](#) and [Application Insights](#). [Azure Monitor](#) also provides centralized monitoring and management for cloud or hybrid solutions.

Aggregate and correlate logs and metrics. A properly instrumented telemetry system provides a large amount of raw performance data and event logs. Make sure that your system processes and correlates telemetry and log data quickly, so that operations staff always has an up-to-date picture of system health. Organize and display data so that you have a cohesive view of problems and can see when events are related to one another.

Consult your corporate retention policy for requirements on how to process data and how long to store data.

Implement automated alerts and notifications. Set up monitoring tools like [Monitor](#) to detect patterns or conditions that indicate potential or current problems. Send alerts to team members who can address problems. Tune the alerts to avoid false positives.

Monitor assets and resources for expirations. Some resources and assets, like certificates, expire. Be sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify your operations team before an asset expires, and escalate the situation if expiration threatens to disrupt applications.

Management

Automate operations tasks. Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Use source control to version code that implements the automation. As with any other code, test your automation tools.

Take an infrastructure-as-code approach to provisioning. Minimize the amount of manual configuration that you need to provision resources. Instead, use scripts and [Azure Resource Manager](#) templates. Keep the scripts and templates in source control, like any other code that you maintain.

Consider using containers. Containers provide a standard package-based interface for deploying applications. When you use containers, you deploy an application by using self-contained packages that include any software, dependencies, and files that you need to run the application. This practice greatly simplifies the deployment process.

Containers also create an abstraction layer between an application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications that run on a host.

Implement resiliency and self-healing. Resiliency is the ability of an application to recover from failures. Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even to another region. For more information, see [Design reliable Azure applications](#). Instrument your applications to report problems immediately so that you can manage outages or other system failures.

Have an operations manual. An operations manual, or *runbook*, documents the procedures and management information that you need for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come into play during a failure or other disruption to your service. Create this documentation during your development process, and keep it up to date afterwards. Treat these resources as living documents that you need to review, test, and improve regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. Your entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

Document on-call procedures. Make sure to document on-call duties, schedules, and procedures, and to share them with all team members. Always keep this information up to date.

Document escalation procedures for third-party dependencies. If your application depends on external third-party services that you don't directly control, you need a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

Use configuration management. Plan configuration changes, make them visible to operations, and record them. You might use a configuration management database or a configuration-as-code approach for these purposes. Audit configuration regularly to ensure that expected settings are actually in place.

Get an Azure support plan and understand the support process. Azure offers many [support plans](#). Determine the right plan for your needs, and make sure that your